Internal Variable Formulation for the Plastic Analysis of Plane Frames

by

Myles Peter Rennie

Thesis presented in partial fulfillment of the requirements for the degree of

Master of Engineering (Civil)

at the University of Stellenbosch

SUPERVISOR:
Prof. W.W. Bird

STELLENBOSCH
November 1999

DECLARATION

I, the undersigned, declare that the work contained in this thesis is my own original work and has not been submitted in its entirety or in part for a degree at any other university.

February 1999

M.P. Rennie

SYNOPSIS

This study provides a comprehensive investigation into the field of advanced analysis of steel portal frames containing truss elements and frame elements with rigid, semi-rigid or pinned connections. Included in the study is the application of advanced software principles in the area of structural mechanics. General problems addressed in mechanics may concern stress analysis, heat conduction, lubrication, electric and magnetic fields, and many others. The focus of this study is in the field of structural analysis.

The advancement of technology is important for the increased efficiency of the analysis process. It leads to a saving of time, which has direct financial benefits and it improves the accuracy of the structural analysis, which results in savings on the cost of the structure. This results in additional pressure on engineers to provide optimally designed structures in less time which in turn leads to an increase in the demand for advanced tools and technology.

The two areas addressed in this study, namely advanced analysis and software technology are dealt with separately at first to investigate and extend the application of each and then they are finally combined in the form of the software package, *Quark*, developed as part of the thesis. Methods to extend the current functionality and accuracy of the analysis algorithms in the structural analysis discipline are investigated and validated. This is followed by a discussion of the theory of Object-Oriented Programming with reference to its application in the field of structural engineering. The philosophy behind, and the features of the software, *Quark* are discussed next. Finally, after some concluding remarks, possible future developments are discussed.

SINOPSIS

Hierdie studie verskaf 'n omvattende ondersoek na die gevorderde analise van staal portaal rame wat beide raam en vakwerk elemente bevat, asook raam elemente met 'rigid', 'semi-rigid' en 'pinned' konneksies. Ingesluit in die studie is die toepassing van gevorderde sagteware beginsels op die gebied van struktuur meganika. Die toepassing van die hierdie meganika sluit in stress analise, hitte oordraging, lubrikasie, elektriese- en magneet velde en vele ander. Die fokus van hierdie studie is slegs in die veld van struktuur analise.

Die vordering van tegnologie is belangrik vir die grooter effektiwiteit behaal in die analise prosess. Dit lei tot 'n besparing van analise tyd, wat 'n direkte besparing van kostes is en die verhoogde akkuraatheid van die ontwerp van strukture wat weereens lei tot 'n besparing in die koste van die struktuur. Hierdie faktore lei daartoe dat grooter eise geplaas word op ontwerp ingenieurs om optimaal ontwerpde strukture in korter tye te verskaf. Dit het die gevolg dat ingenieurs 'n grooter behoefde het aan gevorderde analise toerusting.

Die fokus van hierdie studie naamlik gevorderde struktuur analise en sagteware tegnologie word eerstens afsonderlik behandel om die toepassing van beide te ondersoek en verleng en dan word die twee gekombineer om die resultate in die form van *Quark*, die sagteware ontwikkel as deel van die tesis, te produseer. Metodes om die bestaande struktuur analise algoritmes se funksionaliteit en akkuraatheid te bevorder word ondersoek en voorstelle word gemaak en bewys. Daarna word die teorie agter "Object-Oriented Programming" ondersoek en die toepassings

moontlikhede daarvan op struktuur ingenieurswese voorgestel. Sommige uitstaande funksies *Quark* en die filosofie daaragter word dan bespreek. Na 'n paar afsluitende opmerkings, bespreek die slotsom toekomstige ontwikkelings moontlikhede.

ACKNOWLEDGEMENTS

I would like to express my sincere appreciation to those that have assisted and supported me during this research.

Prof. W.W. Bird, professor at the Department of Civil Engineering of the University of Stellenbosch, who acted as Project Leader of the undertaken study;

Mr. Pieter de Villiers, senior lecturer at the Department of Computer Science of the University of Stellenbosch, who provided me with help, expertise and guidance and for his interest in the progress of this study;

Me. Z Pretorius, who assisted with the typing of sections of this work and for her continuous motivation towards the end of the study;

My parents, Mr. and Me. Myles Rennie and family, for their continuous motivational support and recognition;

To all my friends, for their continuous support and motivation throughout the duration of this study; and

Finally, and most important of all: *TO CHRIST BE THE GLORY*

CONTENTS

Page

LIST OF FIGURES i

LIST OF TABLES ii

CHAPTER 1: Introduction 1
1.1. Overview 1
1.2. Problem Formulation and Object 1
1.3. Organisation of text 2

PART I: THE THEORY OF PLASTIC ANAYLSIS OF PLANE FRAMES 3

CHAPTER 2: Structural Analysis 4
2.1 Design Assumptions and Modelling of Elements 4
2.1.1 Overview 4
2.1.2 Design Assumptions, Provisions and Connection Classifications 5
2.1.3 Modelling of an Elastic Frame Element 9
2.1.4 Modelling of a Elastic Truss Element 12

2.2 Internal Variable formulation for the Plastic Analysis of Rigid- and Semi-Rigid Connection Plane Frames containing Frame and Truss Elements 14
2.2.1 Cross-Section Plastic Strength 14
2.2.2 Formulation of the Structural Problem 15
2.2.3 Incremental Analysis 24
2.2.4 Numerical Analysis 26
2.2.4.1 Single-storey Rectangular Frame 26
2.2.4.2 Two-storey Rectangular Frame 27
2.2.4.3 Two-bay Rectangular Frame 28
2.2.4.4 Rectangular Frame with Distributed Load 29
2.2.4.5 Rectangular Braced Frame 30

PART II: THE THEORY OF OBJECT-ORIENTED SOFTWARE 33

CHAPTER 3: Theory, Design and Implementation of Software 34
3.1 Object oriented programming concepts and implementation 34
3.1.1 Overview 34
3.1.2 Objects and Classes 37
3.1.2.1 Classes 37
3.1.2.2 Access Control 38
3.1.2.3 Constructors and Destructors 38

3.1.2.4 Duality of the term 'Object' 38
3.1.3 Abstraction and Encapsulation 39
3.1.3.1 The Meaning of Abstraction 39
3.1.3.2 The Meaning of Encapsulation 39
3.1.4 Hierarchy 41
3.1.4.1 Hierarchy and complexity 41
3.1.4.2 Hierarchy: Inheritance 41
3.1.4.3 Compatibility of a Base Class and Its Extensions 43
3.1.4.4 Dynamic Binding 44
3.1.4.5 Abstract Classes 44
3.1.4.6 Multiple Inheritance 45
3.1.4.7 Hierarchy: Aggregation 45
3.1.5 Object-Oriented Analysis and Design 47
3.1.5.1 The Meaning of Object-Orientation 47
3.1.5.2 Object-Oriented Analysis 47
3.1.5.3 Object-Oriented Design 47
3.1.5.4 Object-Oriented Programming 48
3.1.5.5 The Available Mechanisms 49
3.1.5.6 Object Behaviour Analysis 49

PART III: THE COMBINATION OF THE THEORIES 51

CHAPTER 4: Designing and Coding of *Quark* 52
4.1 Overview 52
4.2 Graphical Interface and Pre-Processor 53
4.3 Dynamic Numerical Structures and Analysis Algorithms 55
4.4 Post-Processor 56
4.5 Future developments 57
4.5.1 The Graphical Interface 57
4.5.2 The Analysis Interface 58

CHAPTER 5: Conclusions 60

BIBLIOGRAPHY 62

List of Figures

		Page
Figure 1	*Classifications of connections according to Bjorhovde et al. (1990)*	
Figure 2	*Force-displacement relationship of a frame element*	
Figure 3	*Kinematic relationship between local and global displacements of a truss element*	
Figure 5	*Frame element after displacement including plastic and connection rotations*	
Figure 6	*Degrees-of-freedom of truss element*	
Figure 7	*Truss element after displacement*	
Figure 8	*Moment-rotation behaviour of connections*	
Figure 9	*Assumed moment-rotation relationship*	
Figure 10	*Assumed dissipation function*	
Figure 11	*Single-storey frame*	
Figure 12	*Results for single-storey frame*	
Figure 13	*Two-storey frame*	
Figure 14	*Results for two-storey frame*	
Figure 15	*Two-bay frame*	
Figure 16	*Results for two-bay frame*	
Figure 17	*Frame with distributed load*	
Figure 18	*Results for frame with distributed load*	
Figure 19	***B**raced frame*	
Figure 20	*A figure primitive inheritance class hierarchy.*	
Figure 21	*Multiple inheritance in the iostream library*	

List of Tables

Page

Table 1 *Results for single storey frame*

Table 2 *Section properties of two-store frame*

Table 3 *Results for two-storey frame*

Table 4 *Results for two-bay frame*

Table 5 *Results for frame with distributed load*

Table 6 *Braced frame section properties*

Table 7 *Results for braced frame*

Table 8 *Semi-rigid connection properties*

Table 9 *Results for braced frame with the consideration of semi-rigid connections*

CHAPTER 1

INTRODUCTION

1.1 Overview

The rapidly changing needs within the engineering and scientific communities create requirements for faster and more effective solution procedures. Digital computers present a suitable platform from which to provide these procedures.

The solving of complex problems using advanced mathematical procedures and implementing those procedures in computer software presents an ideal topic for a thesis. Providing comprehensive solutions to these problems is an entirely different matter and of necessity, limitations have to be placed on the final solution provided.

The aim of this thesis is to use advanced computer software technology to develop methods and tools that aid and extend tools for advanced plastic analysis of plane frames. The thesis tries to balance the requirements of designers, structural engineers and users of computer software with the functionality and ease-of-use offered my modern programming methods. It is important however, not to overlook the fact that these requirements interface with the process of applying advanced analysis to plane frames, which is in itself a challenge drawing on a number of expert study fields.

The thesis evolved from the basic problem of making existing analysis algorithms as fast and effective as possible and combining them with an ergonomic user interface in order to solve structural analysis problems more easily. A further development was to extend the existing algorithms to include non-rigid nodal connections. This extension introduces another set of variables into the problem and increases the complexity of the solution procedures. A final extension to the original solution procedures was to include *truss* elements. The requirements placed on the solution methods by introducing the *truss* element are very similar those exhibited by a non-rigid nodal connection.

1.2 Problem Formulation and Object-Oriented Philosophy

In this thesis one of the topics to be addressed is Advanced analysis. Advanced analysis refers to any method of analysis that sufficiently represents the strength and stability behaviour such that separate specification member capacity checks are not required. In recent years, there has been an intense interest in the development of advanced analysis methods suitable for design use. However, the current state-of-the-art of advanced analysis is still largely fragmented and disjointed.

Another aspect of the thesis is Object-oriented programming. Object-oriented programming is a software design philosophy, which tries to capture the nature, behaviour and characteristics of real-life object i.e. motor vehicles, structural element or structural connection etc. The software can mimic the behaviour of its real-life partner with some astonishing effect. The effect is of

course subject to the programmers programming skills. To date this technique has been implemented in many academic and commercial applications with great success. In this thesis an attempt has been made to use this technique to its full in order to capture the behaviour of objects such as nodes, members, supports and forces. These objects are then programmed to behave like their real-life partners. The analysis algorithms then reference these objects while the program is running.

The ultimate goal in the future would be to produce an object intelligent enough to completely replicate its real-life partner. This implies that external analysis algorithms will fall way because the behaviour of these objects will handle any simulation. External analysis algorithms are what we now refer to as structural analysis procedures i.e. the setting up and solving of the structural equations. Structures made up of these intelligent objects will have the ability to respond to simulation because each individual objects knows how to behave and react. The combined effect will yield a result, as we now know structural analysis produces. The concept of incorporating the behaviour of these objects with regards to structural analysis is a rapid growing study field, producing new and exciting possibilities for structural analysis.

All of this contributes to make structural analysis a more precise and effective profession. Development and analysis time are reduced considerably and the speed and accuracy at which solutions are produced exceeds anything presented in the past. In this thesis the use of object-oriented technology in advanced analysis will be illustrated and a platform for further developments of its application is provided.

1.3 Organisation of the text

The purpose of this thesis is to introduce recently developed algorithms for the advanced analysis of semi-rigid steel plane frames and the combination of these with the latest in computer software design techniques. The structural issues are discussed in Part I. These include the fundamental principles and theory on which the analysis algorithms are based and these are then combined to provide the final solution algorithms. Part I of the thesis concludes with the inclusion of a section on the numerical analysis to validate the algorithms.

Part II discusses the theory behind the software design techniques implemented and fulfils the second requirement of the thesis. This is a specialised field in computer science and a very broad topic. To ensure that the thesis maintains its balance, this section focuses mainly on the theoretical aspects associated with software design. This section does not attempt to teach the principles discussed, but its aim is to maintain focus on the application of these techniques in the field of structural analysis. Part II is concluded with a brief overview of principles used to implement the theories discussed.

Part III concludes the thesis with the combination of Part I and Part II in the form of *Quark*, the software designed according to the theories discussed in the two parts respectively. Part III does not focus on the use of the software system, but rather describes the thought processes behind the various components of the software designed to finally be combined and form the system called *Quark*. There are many extensions and improvements that can be made to *Quark* to make it more industry robust and better suited for general use. These issues are discussed in the final section on Part III.

PART I: THE THEORY OF PLASTIC ANAYLSIS OF PLANE FRAMES

This section presents the theory for the advanced analysis of rigidly connected plane frames and extends the theory to include semi-rigid connections and truss elements in the structure. Much of the theory is taken directly from Chen and Toma (1994) and from Bird (1995) and is in included here only for the sake of completeness of the arguments presented.

After the overview and statement of design assumptions, the principles for elastic frame and truss elements to be used in the combined model are defined. Next the theory for the semi-rigid connections is described and together with the theory for truss elements, the combined structural model is presented.

The section concludes by describing the analysis method and providing examples of models analysed via the proposed method and a short comparison with other methods.

CHAPTER 2

Structural Analysis

2.1 Introductory theory to the internal variable formulation for the plastic analysis of plane frames

2.1.1 Overview

Today technology has reached such a stage that many behavioural phenomena and structural attributes can easily be considered directly in the analysis. In fact, the rationality of the analysis / design approach may be improved upon in certain instances by accurate analysis of behavioural phenomena which, in current practice, can only be approximated by specification formulas. For complicated frames, second-order inelastic analysis can provide this type of improvement over elastic analysis, for which specified beam-column capacity checks are required. However, for second-order inelastic analysis to be used in design practice, the limitations of different analysis approaches for representing beam-column and connection performance must be clearly identified. In this document we focus on the method derived below. Also, any limit states that are not properly modelled in the analysis must be well understood such that the appropriate checks can be performed according to the design specifications.

Although much work has been done on the analysis and design of rigid framework connections, non-linear behaviour and its effects on the overall frame response requires special attention. In order to implement semi-rigid frame design in practice, engineers need to be assured that they understand the effects of connections on the structure's performance as a whole. To achieve this, it is essential to develop a readily understood analysis / design approach that can provide reliable, economical and safe designs. Also, the designs should provide an economic trade-off when compared with other more conventional design methods. The main obstacles encountered in the design implementation of semi-rigid frames are:

- Classification of connection uncertainties.
- Need for a reliable and general connection moment-rotation model.
- Development of efficient analysis methods.
- Concerns on strength, stability and serviceability limit state conditions.

With computer based analysis techniques in place, future refinements in frame analysis and design will likely focus more on overall system response and less on individual member response. The focus of this thesis is on the development of advanced analysis methods and verification of second-order inelastic analysis for use in design practise. Advanced analysis holds many answers to real behaviour of steel structures.

2.1.2 Design Assumptions, Provisions and Connection Classifications

The steel framework is one of the most commonly used structural systems in modern construction. The analysis of such systems depends largely on the assumptions adopted in the modelling of its element, especially those concerning the behaviour of the beam to column connections. Conventional steel frame analysis methods use two highly idealised connection models: the rigid jointed model and the pin-jointed model. Since the actual behaviour of joints in a frame always fall between these two extremes, much attention has been focussed on a more accurate modelling of such connections.

The proposed analysis is based on the thermodynamic internal variable approach discussed by Martin(1980) and formulated by Bird (1995). Added to this is the three-parameter connection model proposed by Kishi and Chen (1990) to model the moment-rotation response of semi-rigid connection.

The general assumptions used in modelling the beam-column elements are:

- All elements are initially straight and prismatic. Plane cross-sections remain plane after deformation.
- Local buckling and lateral torsion buckling are not considered. Therefore, all members are assumed to be fully compact and adequately braced to preclude out-of-plane deformations.
- Large rigid-body displacements are allowed, but the member deformations and strains are small.
- The formulation is limited by its ability to model plastic hinges only at the element ends.
- Strain hardening due to inelastic rotations is considered here.
- Connection moment-rotation behaviour, as well as semi-rigid connections, is modelled by non-linear rotational springs attached at the element ends.

The assumptions that the member is prismatic and member distortions are small are reasonable for ordinary steel frame structures. Although the steel frame may undergo large rigid-body displacements at collapse, the distortion of each member with respect to its chord length in the displaced configuration will remain small since steel members with compact cross sections usually exhibit high bending rigidity. Large deformation theory is useful to model the full post-collapse behaviour of members in the structure. However, for many types of steel structures, large strains usually do not occur until the members are loaded into the post-collapse region.

Element bowing effects are not considered in the present work because many practical frame members usually have slenderness ratios in the range for which the axial shortening if often dominated by inelastic axial deformation. Maybe however, for very slender beam-column members, the bowing effects ought to be considered in the element stiffness formulation. Alternatively, the element may be broken up into several elements to approximate the member bowing effects.

Inelastic behaviour in the member is assumed to be restricted to a zero-length plastic hinge. The reduction of the plastic moment capacity at the plastic hinge due to the presence of the axial force is considered. Once a plastic hinge is formed, the cross-section forces are allowed to move along the yield surface of the plastic hinge as described in *Section 2.2.1.* The benefits of strain hardening at the plastic hinges are considered. The ability of a beam-column to develop

significant strain hardening is dependant on may factors, such as moment gradient and interaction of local and lateral-torsion buckling effects, and distributed yielding along the member length.

Much research and development work has focused primarily on the analysis and design on frames based on idealisation of the joints as either fully rigid or pinned. In reality, the actual behaviour of the structure is as much dependent on the connection and joint characteristics as on the individual component elements making up the structure. Ample research evidence exists which establishes that the observed joint behaviour is substantially different from the assumed idealised models. Depending on the stiffness, strength, and deformation capacity, the connections in a structural framework can influence the behaviour of the structure in several ways. Under static loads, the connection deformations contribute to the vertical deflections of the beams and the lateral drift of the frame. The moment resistance of the connections will influence the internal force distribution and the local and global stability of the frame.

Realising the potential influence of the connections on frame performance, the American Institute of Steel Construction (AISC, 1986, 1989) has introduced provisions to allow designers to consider explicitly the behaviour of connections in the design of structural steel frames.

The ASD Specification (AISC, 1989) list three types of constructions for designing a multi-story frame:

- Rigid framing. This construction assumes that the beam-to-column connections have sufficient rigidity to maintain the original geometric angle between intersecting members. Rigid connections are assumed for elastic structural analysis.
- Simple framing. This construction assumes that, when the structure is loaded with gravity loads, the beam and girder connections transfer only vertical shear reactions without bending moment. The connections are allowed to rotate freely without any restraint. This type of connection is also called a shear connection.
- Semi-rigid framing. This construction assumes that the connections can transfer vertical shear and also have adequate stiffness and capacity to transfer some moment.

The AISC-LRFD Specifications (1986) designate two types of construction in their provisions: *Type FR (fully restrained)* and *Type PR (partially restrained)*. Type FR corresponds to rigid framing. Type PR includes Simple framing and Semi-rigid framing. If type PR construction is used, the effect of connection flexibility must be considered in the analysis and design of the structure.

To be pertinent, the rigidity of the connection should be defined with respect to the rigidity of the connection member (Colson, 1991). For general application to a wide range of beam-to-column connections, Bjorhovde et al. (1990) introduced a non-dimensional system of classification that compares the connection stiffness to the beam stiffness. In defining the beam stiffness, a reference beam length of 5d is used, where d is the depth to which the connection is attached.

The non-dimensional parameters used in the classification of connections are:

$$\bar{m} = \frac{M}{M_p} \text{ and } \bar{\theta} = \frac{\theta_r}{\theta_p} \qquad (2.1.2.1)$$

in which θ_r is the relative deformation angle of the connection, $\theta_p = \dfrac{M_p}{\left(\dfrac{EI_b}{5d}\right)}$, I_b and L_b are the moment of inertia and the length of the beam, and M_p is the full plastic moment capacity of the beam. The classification is based on the strength and stiffness of the connections with the boundary regions shown in *Figure 1*. The three different regions in *Figure 1* are defined as:

- Rigid connection

  In terms of strength: $\bar{m} \geq 0.7$ *(2.1.2.2)*

  In terms of stiffness: $\bar{m} \geq 2.5\bar{\theta}$ *(2.1.2.3)*

- Semi-rigid connection

  In terms of strength: $0.7 > \bar{m} > 0.2$ *(2.1.2.4)*

  In terms of stiffness: $2.5\bar{\theta} > \bar{m} > 0.5\bar{\theta}$ *(2.1.2.5)*

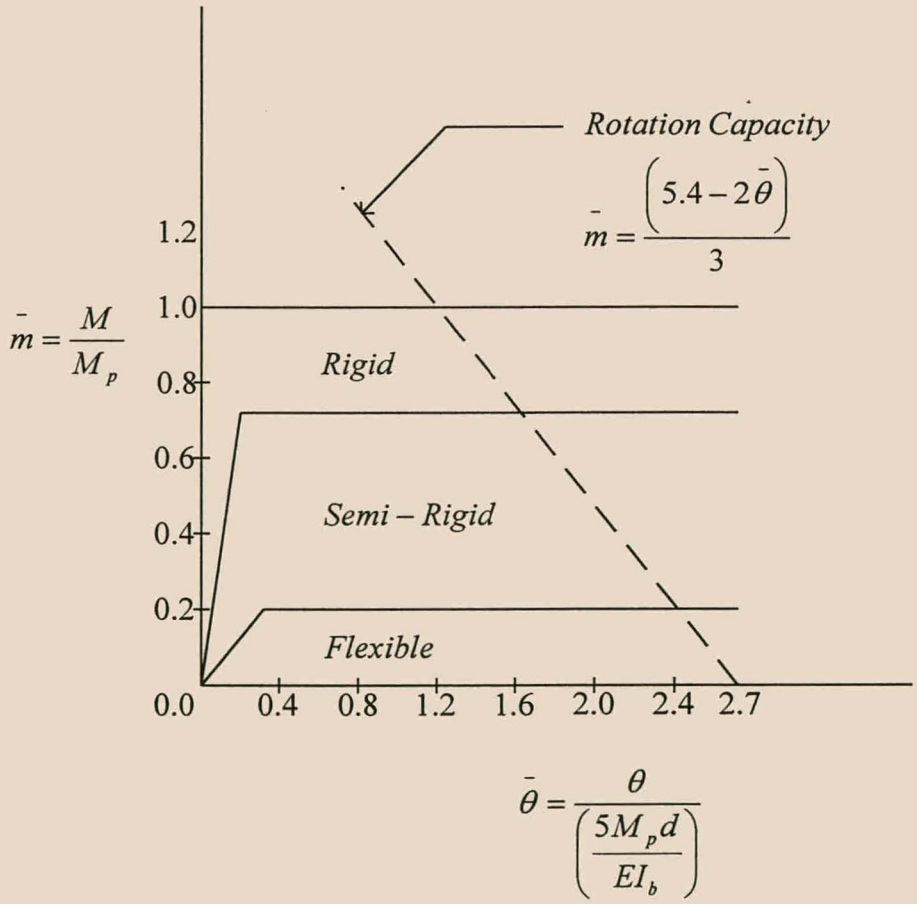

Figure 1 *Classifications of connections according to Bjorhovde et al. (1990)*

- Flexible connection:

In terms of strength: $\bar{m} \leq 0.2$ *(2.1.2.6)*

In terms of stiffness: $\bar{m} \leq 0.5\bar{\theta}$ *(2.1.2.7)*

Bjorhovde et al. (1990) also have proposed an expression for calculating the required capacity of the connection based on a reference beam length and by curve fitting with test data. The simplified expression is written as:

$$\bar{m} = \frac{\left(5.4 - 2\bar{\theta}\right)}{3} \qquad \textit{(2.1.2.8)}$$

According to this formula, the required rotation capacity of the beam-to-column connection depends on the ration of the ultimate moment capacity of the connection to the fully plastic moment of the beam, and it is inversely proportional to the initial connection stiffness, R_{ki}. In other words, the smaller the initial connection stiffness, the larger the necessary rotation capacity. *Equation (2.1.2.8)* is plotted and shown in *Figure 1.* This connection classification system may be used for the selection of connections for use in the analysis and design of semi-rigid frames (Kishi et al., 1992; Liew et al., 1993a and b).

To properly apply the LRFD specification for the design of semi-rigid frames, it is necessary to develop practical means for modelling the moment-rotation behaviour of semi-rigid connections. Also, it is necessary to provide a means for designers to execute the analysis and design quickly and accurately.

2.1.3 Modelling of an Elastic Frame Elements

Consider a prismatic frame element of length L and moment of inertia I with modulus of elasticity E shown in *Figure 2.* The force-displacement relationship of this element may be written as:

$$\begin{Bmatrix} M_A \\ M_B \\ P \end{Bmatrix} = \frac{EI}{L}\begin{bmatrix} S_1 & S_2 & 0 \\ S_2 & S_1 & 0 \\ 0 & 0 & \frac{A}{I} \end{bmatrix}\begin{Bmatrix} \theta_A \\ \theta_B \\ e \end{Bmatrix} \qquad (2.1.3.1)$$

In which $M_A, M_B, \theta_A, \theta_B$ are the end moments and the corresponding joint rotations at element end A and B respectively. P, e (Positive in tension) are the axial force and displacement in the longitudinal direction of the element. S_1 and S_2 are the stability functions that account for the effect of the axial force on the bending stiffness of the member. The conventional stability functions can be written as:

$$S_1 = \frac{\pi\sqrt{\rho}\sin(\pi\sqrt{\rho}) - \pi^2\rho\cos(\pi\sqrt{\rho})}{2 - 2\cos(\pi\sqrt{\rho}) - \pi\sqrt{\rho}\sin(\pi\sqrt{\rho})} \quad if\ P < 0 \qquad (2.1.3.2)$$

$$S_1 = \frac{\pi^2\rho\cosh(\pi\sqrt{\rho}) - \pi\sqrt{\rho}\sinh(\pi\sqrt{\rho})}{2 - 2\cosh(\pi\sqrt{\rho}) + \pi\sqrt{\rho}\sinh(\pi\sqrt{\rho})} \quad if\ P > 0 \qquad (2.1.3.3)$$

$$S_2 = \frac{\pi^2\rho - \pi\sqrt{\rho}\sin\rho(\pi\sqrt{\rho})}{2 - 2\cos(\pi\sqrt{\rho}) - \pi\sqrt{\rho}\sin(\pi\sqrt{\rho})} \quad if\ P < 0 \qquad (2.1.3.4)$$

$$S_2 = \frac{\pi\sqrt{\rho}\sinh(\pi\sqrt{\rho}) - \pi^2\rho}{2 - 2\cosh(\pi\sqrt{\rho}) + \pi\sqrt{\rho}\sinh(\pi\sqrt{\rho})} \quad if\ P > 0 \qquad (2.1.3.5)$$

Where $\rho = \frac{P}{\left(\frac{\pi^2 EI}{L^2}\right)}$, and P is taken as positive in tension. It is clear from *Equations (2.1.3.2)* to *(2.1.3.5)* that the numerical solution obtained from these equations are indeterminate when the axial force is equal to zero. The circumvent this problem and to avoid the use of different equations of S_1 and S_2 for a different sign of axial forces, Goto and Chen (1987) have proposed a set of expressions that make use of the power-series expansions to approximate the stability functions. The power series has been shown to converge to a high degree of accuracy within the first ten terms of the series expansion. Alternatively, if the axial force in the member falls within the range

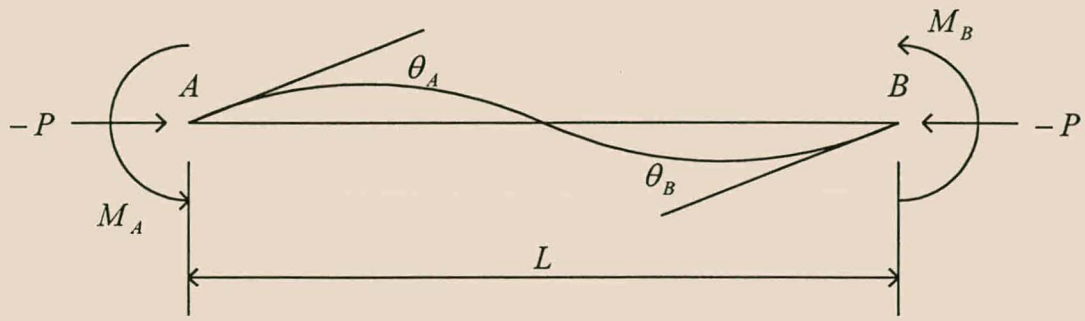

Figure 2 *Force-displacement relationship of a frame element*

$-2.0 \le \rho \le 2.0$, the following simplified expressions may be used to closely approximate the stability functions (Lui, 1985):

$$S_1 = 4 + \frac{2\pi^2\rho}{15} - \frac{(0.01\rho + 0.543)\rho^2}{4+\rho} - \frac{(0.004\rho + 0.285)\rho^2}{8.183+\rho} \qquad (2.1.3.6)$$

$$S_2 = 2 - \frac{\pi^2\rho}{30} + \frac{(0.01\rho + 0.543)\rho^2}{4+\rho} - \frac{(0.004\rho + 0.285)\rho^2}{8.183+\rho} \qquad (2.1.3.7)$$

Equations (2.1.3.6) and *(2.1.3.7)* are applicable for members in tension (positive P) and compression (negative P). For most practical applications, *Equations (2.1.3.6)* and *(2.1.3.7)* give and excellent correlation to the 'exact' expressions given by *Equations (2.1.3.2)* to *(2.1.3.5)*. However, for ρ other than the range $-2.0 \le \rho \le 2.0$, the conventional stability functions should be used instead. The stability function approach enables the use of only element for each frame member and still maintains a good accuracy in the element stiffness terms and in the force recovery process.

The element stiffness relationship from *Equation (2.1.3.1)* my be written symbolically as

$$\boldsymbol{f}^{el} = \boldsymbol{K}^{el}\boldsymbol{d}^{el} \qquad (2.1.3.8)$$

in which $\boldsymbol{f}^{el}, \boldsymbol{d}^{el}$ is the local element end forces and displacements, respectively, and $\boldsymbol{K}^{el}$ is the local element basic stiffness matrix. For a plane frame member, three additional degrees of freedom are required to describe the total displacement of the member. If d_{gl}, $dg2$... and d_{g6} are defined as the global translation and rotational degrees of freedom of a frame member, it can be shown that the local displacements are related to the global displacements by

$$d_{c1} = \theta_A = \theta_0 + d_{g3} - \tan^{-1}\frac{y_0 + d_{g5} - d_{g2}}{x_0 + d_{g4} - d_{g1}} \qquad (2.1.3.9)$$

$$d_{c2} = \theta_B = \theta_0 + d_{g6} - \tan^{-1}\frac{y_0 + d_{g5} - d_{g2}}{x_0 + d_{g4} - d_{g1}} \qquad (2.1.3.10)$$

$$d_{c3} = \frac{(2x_0 + d_{g4} - d_{g1})(d_{g4} - d_{g1}) + (2y_0 + d_{g5} - d_{g2})(d_{g5} - d_{g2})}{L_f + L} \qquad (2.1.3.11)$$

The expression for d_{c3} in *Equation (2.1.3.11)* is more accurate than the value calculated from $L_f - L$. This is because *Equation (2.1.3.11)* avoids finding the small difference between large member lengths (Cook et al., 1989). *Equation (2.1.3.11)* is obtained by writing $d_{c3} = \left(L_f{}^2 - L^2\right)/\left(L_f + L\right)$, and then solving for d_{c3}. In the denominator, $L_f + L \approx 2L$ may be used, since the small displacement theory is presumed from the corotational chord element (Belytschko and Hsieh, 1973).

$$\boldsymbol{d}^{el} = \left[\theta_A \; \theta_B \; e\right]^T \qquad (2.1.3.12)$$

$$\boldsymbol{d}^{e} = \left[d_{g1} \; d_{g2} \; d_{g3} \; d_{g4} \; d_{g5} \; d_{g6}\right]^T \qquad (2.1.3.13)$$

$$\boldsymbol{T}_{cg} = \begin{bmatrix} -s/L & c/L & 1 & s/L & -c/L & 0 \\ -s/L & c/L & 0 & s/L & -c/L & 1 \\ -c & -s & 0 & c & s & 0 \end{bmatrix} \qquad (2.1.3.14)$$

in which $c = \cos\theta$, $s = \sin\theta$, and θ is the angle of inclination of the chord of the deformed member. Based on the principle of equilibrium, the forces in the two systems are related by

$$\boldsymbol{f}^{e} = \boldsymbol{T}_{cg}{}^{T} \boldsymbol{f}^{el} \qquad (2.1.3.15)$$

Substituting $\boldsymbol{f}^{el}$ from *Equation (2.1.3.8)* into *Equation (2.1.3.15)* gives the following expression for $\boldsymbol{f}^{e}$, the element global end forces

$$\boldsymbol{f}^{e} = \boldsymbol{T}^{T}{}_{cg} \boldsymbol{K}^{el} \boldsymbol{T}_{cg} \boldsymbol{d}^{e} \qquad (2.1.3.16)$$

where

$$\boldsymbol{d}^{e} = \boldsymbol{T}_{cg} \boldsymbol{d}^{el} \qquad (2.1.3.17)$$

due once again to the principle of equilibrium. *Equation (2.1.3.16)* is the force-displacement relationships of a frame element in the global coordinate system, and it may be written symbolically as

$$\boldsymbol{f}^{e} = \boldsymbol{K}^{e} \boldsymbol{d}^{e} \qquad (2.1.3.18)$$

where $\boldsymbol{K}^{e}$ represents the stiffness matrix of a frame element in global coordinates. It should be noted that the derivation of the stiffness matrix, $\boldsymbol{K}^{e}$, the joints are assumed to be rigid. If plastic hinges or connections are present at the element ends, the tangent stiffness matrix needs to be modified. These modifications are discussed in *Section 2.2.2.*

2.1.4 Modelling of an Elastic Truss Element

Trusses play a vital role when trying to enhance the lateral-load resistance of a structure. In design trusses are assumed to carry axial force only. Therefore it is justifiable to use truss elements to model bracing elements. Truss elements may also be used for modelling gravity columns that do not participate in the lateral-force resisting system. These gravity columns which are widely used in many types of low-rise industrial buildings and tall office building frames (Springfield, 1991), are usually designed to carry only gravity loads.

The stiffness relationship for a bracing element can be obtained from the stiffness relationship of a frame element by deleting the appropriate rows and columns in *Equation (2.1.3.16)* that correspond to the rotational degrees of freedom of the element. The resulting stiffness relationship of a truss element is:

$$\boldsymbol{f}^e = \boldsymbol{T}^T{}_{cg}\boldsymbol{K}^{el}\boldsymbol{T}_{cg}\boldsymbol{d}^e = \boldsymbol{K}^e\boldsymbol{d}^e \qquad (2.1.4.1)$$

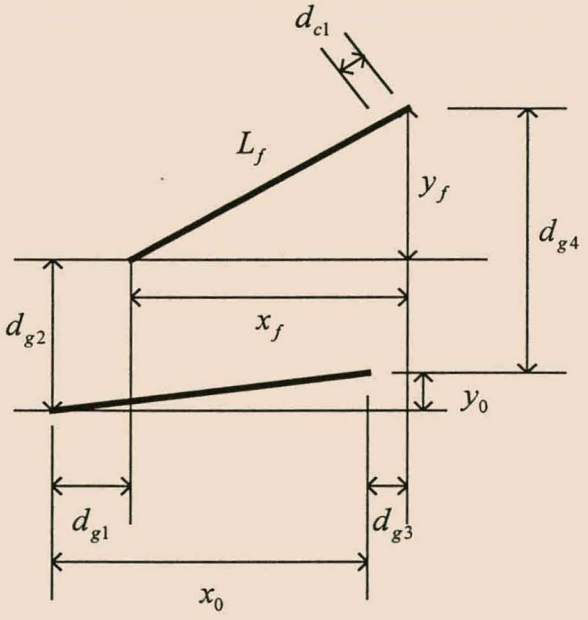

Figure 3 *Kinematic relationship between local and global displacements of a truss element*

where (refer to *Figure 3*)

$$\boldsymbol{f}^e = \left[f_{g1}\, f_{g2}\, f_{g3}\, f_{g4} \right]^T \qquad (2.1.4.2)$$

$$\boldsymbol{d}^e = \left[d_{g1}\, d_{g2}\, d_{g3}\, d_{g4} \right]^T \qquad (2.1.4.3)$$

$$\boldsymbol{T}_{cg} = \left[-\cos\theta \; -\sin\theta \; \cos\theta \; \sin\theta \right] \qquad (2.1.4.4)$$

$$\boldsymbol{K}^{el} = \frac{EA}{L} \qquad (2.1.4.5)$$

in which $s = \sin\theta$ and $c = \cos\theta$. θ is the inclination angle of the displaced element chord.

2.2 Internal Variable formulation for the Plastic Analysis of Rigid- and Semi–Rigid Connection Plane Frames

2.2.1 Cross-Section Plastic Strength

Frame elements are assumed to remain elastic until the second-order forces at the critical location in the element reach the cross section plastic strength. Once the plastic strength is reached, a plastic hinge is formed and the cross-section behavior is assumed to be perfectly plastic.

The AISC-LRFD bilinear interaction equations (AISC-LRFD, 1986) for a member of compact cross-section and of zero length are used in the present formulation for elastic-plastic hinge analysis of members subjected to strong- or weak-axis bending. These equations are written as:

$$\frac{P}{P_y}+\frac{8}{9}\frac{M}{M_p}=1.0 \quad for \ \frac{P}{P_y}\geq 0.2 \qquad (2.2.1.1)$$

$$\frac{P}{2P_y}+\frac{M}{M_p}=1.0 \quad for \ \frac{P}{P_y}<0.2 \qquad (2.2.1.2)$$

where P_y is the squash load, M_p is the plastic moment capacity for the member under pure bending action, and P and M are the second-order axial force and bending moment at the cross-section being considered. *Equations 2.2.1.1* and *2.2.1.2* assume the same functional relationship for both strong- and weak-axis strengths and provide a reasonable lower-bound fit to most of the strong-axis strengths, but they are rather conservative for the weak-axis strength when the bending moment and axial forces are dominate.

To incorporate strain hardening ($s \geq 0$) into the cross-sectional plastic strength formulations, *Equations 2.2.1.1* and *2.2.1.2* are adjusted as follow:

$$\frac{P}{P_y}+\frac{8}{9}\frac{M}{M_p}=1.0+s\phi \quad for \ \frac{P}{P_y}\geq 0.2 \qquad (2.2.1.3)$$

$$\frac{P}{2P_y}+\frac{M}{M_p}=1.0+s\phi \quad for \ \frac{P}{P_y}<0.2 \qquad (2.2.1.4)$$

as described in *Section 2.2.2* with *Equation 2.2.2.27* and *2.2.2.28*. Here ϕ is the plastic rotation at the element end. In the case of strain softening ($s < 0$) *Equations 2.2.1.3* and *2.2.1.4* follow the formulation specified by *Equation 2.2.2.28. Equations 2.2.1.3* and *2.2.1.4* are used in *Section 2.2.3* for the calculation of the Moment (M) at the element ends.

2.2.2 Formulation of the structural problem

The proposed formulation is the thermodynamically based internal variable formulation discussed by Martin. To incorporate the effect of connection flexibility into the element stiffness relationships, the connection is modeled as a rotational spring with the moment-rotation relationship described by *Equations (2.2.2.17)* or *(2.2.2.18)*. The displacement of the structure, which is in this case a plane frame, can be represented by the displacement vector $\boldsymbol{u}$. Plastic hinges i.e. internal slips are represented by the components of the vector $\boldsymbol{\phi}$. The presence of the connections introduces relative rotations of the element ends i.e. semi-rigid connection rotations are represented by the components of the vector $\boldsymbol{\delta}$.

Figure 4 *Degrees-of-freedom of frame element*

The six local degrees-of-freedom of the single frame element is depicted in *Figure 4*. Using this single frame element allows us to identify the vectors $\boldsymbol{u}$, $\boldsymbol{\phi}$ and $\boldsymbol{\delta}$. The element is located between two nodes i and j each with three global degrees of freedom. The displaced element shown in *Figure 5* allows us to identify the plastic rotations and connection rotations. The respective plastic rotations are shown as ϕ_1 and ϕ_2 and the connection rotations as δ_1 and δ_2. Notice that the element end rotations include the elastic rotations and the end connection rotation or relative rotations.

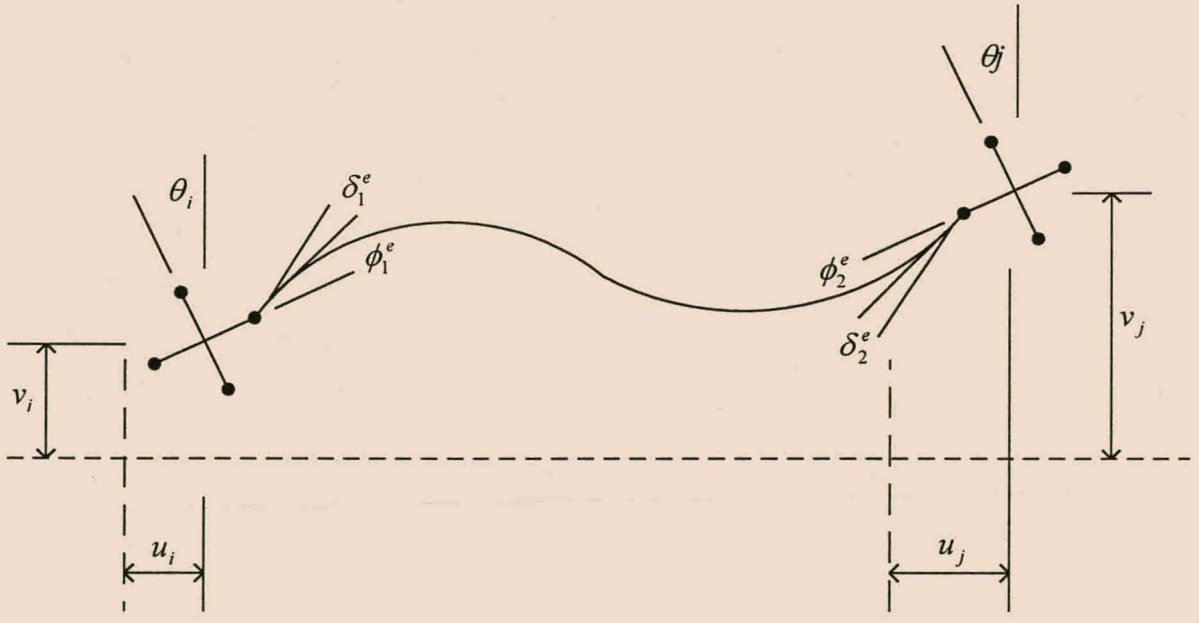

Figure 5 *Frame element after displacement including plastic and connection rotations*

The element displacement vector $\boldsymbol{u}^e$ is defined as,

$$\boldsymbol{u}^e = \begin{Bmatrix} u_1^e \\ v_1^e \\ \theta_1^e \\ u_2^e \\ v_2^e \\ \theta_2^e \end{Bmatrix} \qquad (2.2.2.1)$$

In terms of global or nodal displacements, the element displacement vector becomes

$$\boldsymbol{u}^e = \begin{Bmatrix} u_1^e \\ v_1^e \\ \theta_1^e - \delta_1^e + \phi_1^e \\ u_2^e \\ v_2^e \\ \theta_2^e - \delta_2^e + \phi_2^e \end{Bmatrix} \qquad (2.2.2.2)$$

The nodal and element contributions to the element displacements are separated to make it more convenient as follows

$$\boldsymbol{u}^e = \begin{Bmatrix} u_i \\ v_i \\ \theta_i \\ u_j \\ v_j \\ \theta_j \end{Bmatrix} - \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{Bmatrix} \delta_1^e \\ \delta_2^e \end{Bmatrix} + \begin{bmatrix} 0 & 0 \\ 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix} \begin{Bmatrix} \varphi_1^e \\ \varphi_2^e \end{Bmatrix} \qquad (2.2.2.3)$$

From this equation and *Figure 5* it is quite clear that the relative rotations have to be subtracted from the positive nodal elastic rotation to obtain the net nodal elastic rotation. It can also be seen that if the relative rotations were zero i.e. rigid end connections, the element displacement vector would reduce to that proposed by Bird (1995). The element displacement vector can now be written as

$$\boldsymbol{u}^e = \begin{Bmatrix} \boldsymbol{u}_i \\ \boldsymbol{u}_j \end{Bmatrix} - \boldsymbol{N}^e \boldsymbol{\delta}^e + \boldsymbol{N}^e \boldsymbol{\varphi}^e \qquad (2.2.2.4)$$

From this the element strain energy, F^e can be computed as

$$F^e = \frac{1}{2} \boldsymbol{u}^{eT} \boldsymbol{K}^e \boldsymbol{u}^e \qquad (2.2.2.5)$$

where $\boldsymbol{K}^e$ is the standard elastic element stiffness matrix for a frame element as calculated in *Section 2.1.3* and *2.1.4*. At this point it is necessary to introduce the truss element and the truss displacement vector.

For structures subjected to lateral wind or earthquake loading, truss diagonal bracings may be used to reduce the frame drifts and to enhance the lateral-load resistance of the structure. In design, the braces are usually assumed to carry axial forces only. Therefore it is justifiable to use truss elements to model the bracing members.

Figure 6 *Degrees-of-freedom of truss element*

The truss element is once again modelled between two nodes *i* and *j* with four degrees of freedom, as depicted in *Figure 6*. In *Figure 7* it shows an element in its deformed state with all displacements in their positive sense. Truss elements do not accommodate rotational degrees-of-freedom. This implies that truss element can be implemented into the current formulation without altering the accepted standard truss. Bearing this in mind, it does take some effort though to incorporate the truss element into the current formulation because of the nature of the solution process.

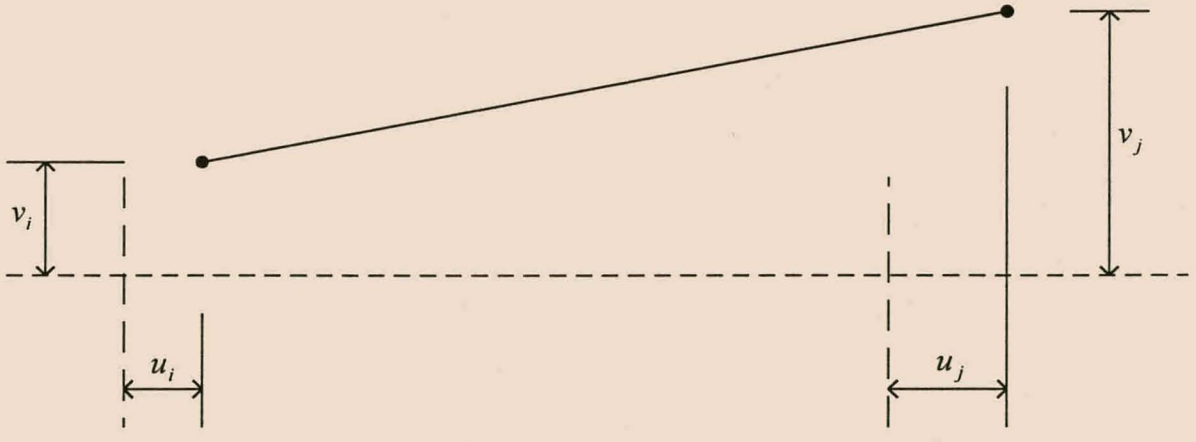

Figure 7 *Truss element after displacement*

We then define the truss element displacement vector as

$$\boldsymbol{u}^e = \begin{Bmatrix} u_1^e \\ v_1^e \\ u_2^e \\ v_2^e \end{Bmatrix} \qquad (2.2.2.6)$$

which can be written as

$$\boldsymbol{u}^e = \begin{Bmatrix} \boldsymbol{u}_i \\ \boldsymbol{u}_j \end{Bmatrix} \tag{2.2.2.7}$$

Caution should be taken not to confuse this displacement vector with that of the frame element. The reason for using this notation is to simplify the writing of the solution algorithm employed. The frame displacement vector will have a small *f* added i.e. $\boldsymbol{u}_f$ and the truss displacement vector will have a small *t* indicating that it is a truss element i.e. $\boldsymbol{u}_t$.

Using this convention the strain energy for a frame element can be written, after substituting the value of $\boldsymbol{u}_f^e$ as follows

$$F^e = \frac{1}{2}\begin{Bmatrix} \mathbf{u}_i \\ \mathbf{u}_j \end{Bmatrix}^T \mathbf{K}^e \begin{Bmatrix} \mathbf{u}_i \\ \mathbf{u}_j \end{Bmatrix} + \frac{1}{2}\delta^{eT}\mathbf{N}^{eT}\mathbf{K}^e\mathbf{N}^e\delta^e + \frac{1}{2}\varphi^{eT}\mathbf{N}^{eT}\mathbf{K}^e\mathbf{N}^e\varphi^e - \begin{Bmatrix} \mathbf{u}_i \\ \mathbf{u}_j \end{Bmatrix}^T \mathbf{K}^e\mathbf{N}^e\delta^e + \begin{Bmatrix} \mathbf{u}_i \\ \mathbf{u}_j \end{Bmatrix}^T \mathbf{K}^e\mathbf{N}^e\varphi^e - \varphi^{eT}\mathbf{N}^{eT}\mathbf{K}^e\mathbf{N}^e\delta^e \tag{2.2.2.8}$$

$$F^e = \frac{1}{2}\begin{Bmatrix} \boldsymbol{u}_i \\ \boldsymbol{u}_j \end{Bmatrix}^T \boldsymbol{K}^e \begin{Bmatrix} \boldsymbol{u}_i \\ \boldsymbol{u}_j \end{Bmatrix} + \frac{1}{2}\delta^{eT}\boldsymbol{H}^e\delta^e + \frac{1}{2}\varphi^{eT}\boldsymbol{H}^e\varphi^e - \begin{Bmatrix} \boldsymbol{u}_i \\ \boldsymbol{u}_j \end{Bmatrix}^T \boldsymbol{L}^e\delta^e + \begin{Bmatrix} \dot{\boldsymbol{u}}_i \\ \boldsymbol{u}_j \end{Bmatrix}^T \boldsymbol{L}\varphi^e - \varphi^{eT}\boldsymbol{H}^e\delta^e \tag{2.2.2.9}$$

where $\boldsymbol{L}^e = \boldsymbol{K}^e\boldsymbol{N}^e$ and $\boldsymbol{H}^e = \boldsymbol{N}^{eT}\boldsymbol{K}^e\boldsymbol{N}^e$. The strain energy for a truss element can simply be written as

$$F^e = \frac{1}{2}\begin{Bmatrix} \boldsymbol{u}_i \\ \boldsymbol{u}_j \end{Bmatrix}^T \boldsymbol{K}^e \begin{Bmatrix} \boldsymbol{u}_i \\ \boldsymbol{u}_j \end{Bmatrix} \tag{2.2.2.10}$$

where the displacement vector is again that of the truss element.

The total strain energy *F* in the structure is the sum of all the element contributions and is a homogeneous quadratic function in terms of the global displacement vector $\boldsymbol{u}$, for trusses and frames, the vector of plastic hinges ϕ for frame elements and the relative rotations allowing for the semi-rigid end connections δ, for frame elements. This can be expressed as

$$F = \frac{1}{2}\boldsymbol{u}^T\boldsymbol{K}\boldsymbol{u} + \frac{1}{2}\delta^T\boldsymbol{H}\delta + \frac{1}{2}\varphi^T\boldsymbol{H}\varphi - \boldsymbol{u}^T\boldsymbol{L}\delta + \boldsymbol{u}^T\boldsymbol{L}\varphi - \varphi^T\boldsymbol{H}\delta \tag{2.2.2.11}$$

Using the methods proposed by Kishi and Chen (1990) in their three-parameter model, the values of the relative rotations for the semi-rigid connections can be calculated. The generalised equation of the three-parameter model has the form

$$m = \frac{\delta}{\left(1+\delta^n\right)^{\frac{1}{n}}} \text{ for } \delta > 0 \text{ and } m > 0 \qquad (2.2.2.12)$$

or equivalently,

$$\delta = \frac{m}{\left(1-m^n\right)^{\frac{1}{n}}} \text{ for } \delta > 0 \text{ and } m > 0 \qquad (2.2.2.13)$$

The parameters in these equations are defined as:

$$m = \frac{M}{M_u} \qquad (2.2.2.14)$$

$$\delta = \frac{\delta_r}{\delta_o} \qquad (2.2.2.15)$$

$\delta_o = \frac{M_u}{R_{ki}}$, the reference plastic rotation (2.2.2.16)

M_u = ultimate moment capacity of the connection

R_{ki} = initial connection stiffness

n = shape parameter

δ_r = any arbitrary rotation

The connection tangent stiffness R_{kt} at an arbitrary rotation $|\delta_r|$ can be evaluated by differentiating M with respect to $|\delta_r|$, and it is expressed as

$$R_{kt} = \frac{dM}{d\,|\delta_r|_{|\delta_r|}} = \frac{M_u}{\delta_o\left(1-\delta^n\right)^{1+\frac{1}{n}}} \qquad (2.2.2.17)$$

when the connection is loaded, and it is

$$R_{kt} = \frac{dM}{d\,|\delta_r|_{|\delta_r=0|}} = \frac{M_u}{\delta_o} = R_{ki} \qquad (2.2.2.18)$$

when the connection is unloaded.

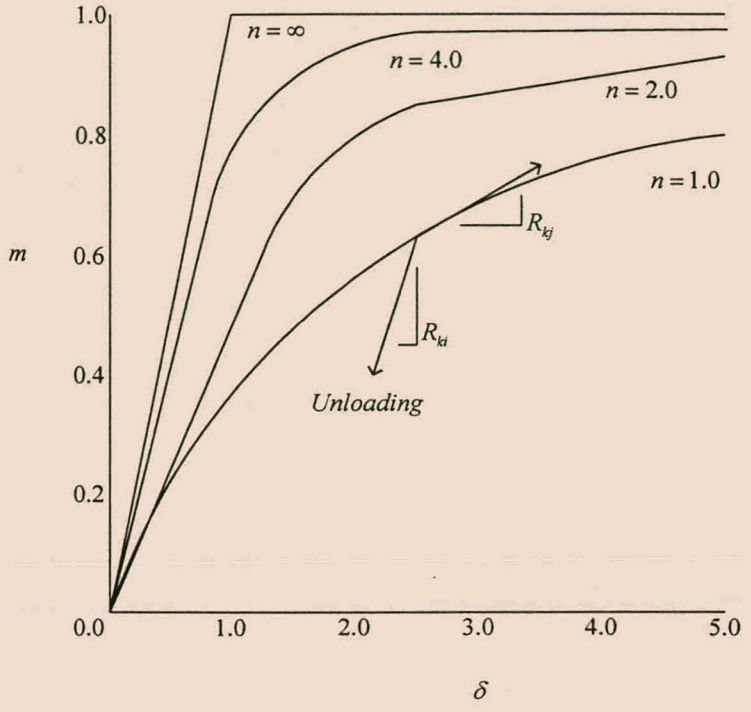

Figure 8 *Moment-rotation behaviour of connections*

Equation (2.2.2.12) and *(2.2.2.13)* has the shape shown in *Figure 8.* The principle merit of this model is that it allows the designer to execute the non-linear structural analysis quickly and accurately. This is because the connection stiffness and the relative rotation can be determined directly from *Equations (2.2.2.13 to 2.2.2.18)* without iteration. Also, this model is based on connection parameters that can be determined analytically based on the connection configuration, thus making it more appropriate for practical use Liew[2] (1992).

With the determination of the relative rotations, the values can be substituted into the respective element strain energy equations to simplify these to the form proposed by Bird (1995), and can be expressed as

$$F^e = \frac{1}{2}\begin{Bmatrix} \boldsymbol{u}_i \\ \boldsymbol{u}_j \end{Bmatrix}^T \boldsymbol{K}^e \begin{Bmatrix} \boldsymbol{u}_i \\ \boldsymbol{u}_j \end{Bmatrix} + \boldsymbol{V}^e + \frac{1}{2}\varphi^{eT}\boldsymbol{H}^e\varphi^e - \begin{Bmatrix} \boldsymbol{u}_i \\ \boldsymbol{u}_j \end{Bmatrix}^T \boldsymbol{Q}^e + \begin{Bmatrix} \boldsymbol{u}_i \\ \boldsymbol{u}_j \end{Bmatrix}^T \boldsymbol{L}^e\varphi^e - \varphi^{eT}\boldsymbol{T}^e \qquad (2.2.2.19)$$

where $\boldsymbol{Q}^e = \boldsymbol{L}^e\delta^e$ and $\boldsymbol{T}^e = \boldsymbol{H}^e\delta^e$ and $\boldsymbol{V}^e = \frac{1}{2}\delta^{eT}\boldsymbol{H}^e\delta^e$.

Once again the total strain energy F in the structure is the sum of all the element contributions, and can be expressed as

$$F = \frac{1}{2}\boldsymbol{u}^T\boldsymbol{K}\boldsymbol{u} + \boldsymbol{V} + \frac{1}{2}\varphi^T\boldsymbol{H}\varphi - \boldsymbol{u}^T\boldsymbol{Q} + \boldsymbol{u}^T\boldsymbol{L}\varphi - \varphi^T\boldsymbol{T} \qquad (2.2.2.20)$$

Differential changes in the kinematic variable yield

$$dF = \frac{\partial F}{\partial \boldsymbol{u}} d\boldsymbol{u} + \frac{\partial F}{\partial \varphi} d\varphi = \boldsymbol{r} d\boldsymbol{u} - \boldsymbol{x} d\varphi \qquad (2.2.2.21)$$

and we identify the internal nodal forces **r** and the internal forces or conjugate forces **x** acting on the hinges. We set

$$\boldsymbol{x} = -\frac{\partial F}{\partial \varphi} \qquad (2.2.2.22)$$

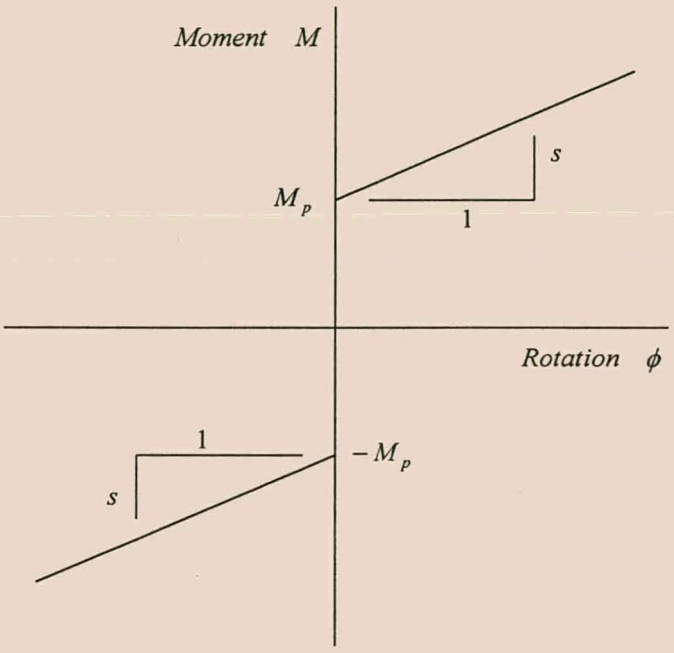

Figure 9 *Assumed moment-rotation relationship*

in order to obtain the forces (moments) applied by the structure to the hinges rather than the forces applied by the hinges to the structure. It follows that both $\boldsymbol{r}$ and $\boldsymbol{x}$ are homogeneous linear functions of $\boldsymbol{u}$ and ϕ

$$\boldsymbol{r} = \boldsymbol{K}\boldsymbol{u} + \boldsymbol{L}\varphi - \boldsymbol{Q} \qquad (2.2.2.23)$$

$$-\boldsymbol{x} = \boldsymbol{L}^T\boldsymbol{u} + \boldsymbol{H}\varphi - \boldsymbol{T} \qquad (2.2.2.24)$$

Equilibrium requires that the internal forces at the nodes are equal to the external loads, $\boldsymbol{p}$, applied at the nodes and therefore we can rewrite *Equation (2.2.2.24)* as

$$\boldsymbol{K}\boldsymbol{u} + \boldsymbol{L}\varphi = \boldsymbol{p} + \boldsymbol{Q} \qquad (2.2.2.25)$$

A rigid plastic relation, as shown in *Figure 9*, governs the rate of change of plastic rotations at hinges. We assume the existence of a dissipation function D, such that

$$\boldsymbol{x} = \frac{\partial D}{\partial \dot{\varphi}} \qquad (2.2.2.26)$$

D is convex in the cases of perfect plasticity ($s = 0$) and hardening ($s \geq 0$) with $D \geq 0$ for all cases and $D = 0$ if and only if $\dot{\varphi} = 0$. It follows that the derivatives of D are discontinuous at the origin, and that, since D will generally be the sum of independent dissipation functions associated with individual hinges, derivatives of D may be discontinuous along lines which are radial in the $\dot{\varphi}$ space. *Figure 10* depicts the contribution of an individual hinge to the total dissipation.

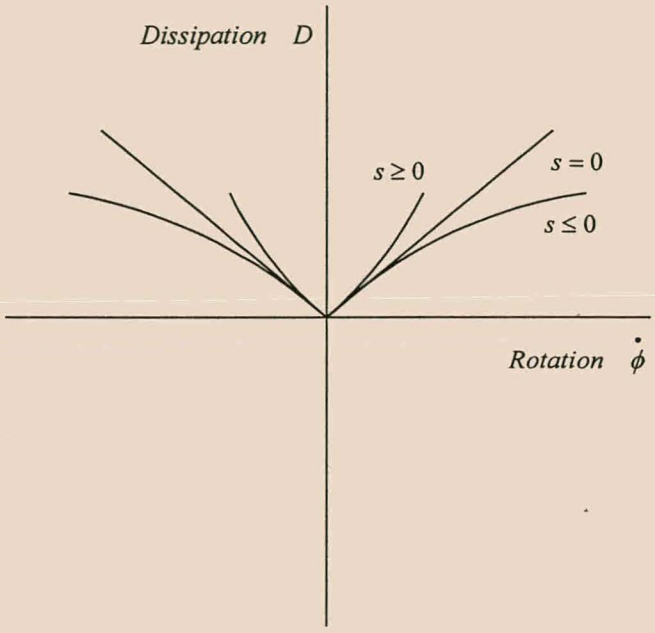

Figure 10 *Assumed dissipation function*

Identifying the components of $\boldsymbol{x}$ as the moments at hinges (i.e. positions in the structure where the moments M have actually exceeded the plastic yield moment), we have for the positive components with $M > 0$ and $\phi > 0$ that

$$M = M_p + s\varphi \qquad (2.2.2.27)$$

and for the negative components with $M < 0$ and $\phi < 0$ that

$$M = -M_p + s\varphi \qquad (2.2.2.28)$$

where s is a linear hardening ($s \geq 0$) or softening ($s < 0$) parameter as indicated in *Figure 10*.

We can thus write the components of $\boldsymbol{x}$ as

$$X_i = \pm M_p + s\int_0 \dot{\varphi} dt \qquad (2.2.2.29)$$

where t denotes time.

If we regard the load applied to the structure as a function of time, $\boldsymbol{p} = \boldsymbol{p}(t)$, $\boldsymbol{u} = \boldsymbol{u}(t)$, $\varphi = \varphi(t)$ and $\dot{\varphi} = \frac{d\varphi}{dt}$. Since the problems are rate independent, the parameter t measures the order of events, rather than real time. The initial condition $\phi(0)$ will be taken to be zero indicating there are no plastic hinge rotations at the start of the analysis. The governing equations can be written as

$$\boldsymbol{K}\boldsymbol{u}(t) + \boldsymbol{L}\varphi(t) = \boldsymbol{p}(t) + \boldsymbol{Q}(t) \qquad (2.2.2.30)$$

$$\boldsymbol{L}^T\boldsymbol{u}(t) + \boldsymbol{H}\varphi(t) = -\left\{\frac{\partial D}{\partial \dot{\varphi}}\right\}_{\dot{\varphi}(t)} + \boldsymbol{T}(t) \qquad (2.2.2.31)$$

2.2.3 Incremental Analysis

In order to set up a numerical procedure which determines the response of the structure to a given load program $p(t)$, we need to divide the time domain into discrete intervals Δt which are not necessarily equal. We seek to satisfy *Equations (2.2.2.31)* and *(2.2.2.32)* only at the end of these intervals. Thus at time t_n, after n intervals have elapsed, these equations are written as

$$\boldsymbol{K}\boldsymbol{u}_n + \boldsymbol{L}\varphi_n = \boldsymbol{p}_n + \boldsymbol{Q}_n \qquad (2.2.3.1)$$

$$\boldsymbol{L}^T\boldsymbol{u}_n + \boldsymbol{H}\varphi_n = -\left\{\frac{\partial D}{\partial \dot{\varphi}}\right\}_{\dot{\varphi}_n} + \boldsymbol{T}_n = -\boldsymbol{x}_n + \boldsymbol{T}_n \qquad (2.2.3.2)$$

In order to trace the behaviour of the structure under varying load, we introduce a series of steps t_1, t_2, t_3 .... Assuming that at step t_{n-1} we have a solution to *Equations (2.2.3.1)* and *(2.2.3.2)*, we now seek a solution at step t_n. Using the relationship

$$\varphi_n = \varphi_{n-1} + \Delta\varphi \qquad (2.2.3.3)$$

we can rewrite *Equation (2.2.3.1)* as

$$\boldsymbol{K}\boldsymbol{u}_n + \boldsymbol{L}\varphi_{n-1} + \boldsymbol{L}\Delta\varphi = \boldsymbol{p}_n + \boldsymbol{Q}_n \qquad (2.2.3.4)$$

and hence we can express the displacement as

$$\boldsymbol{u}_n = \boldsymbol{u}_n^E - \boldsymbol{K}^{-1}\boldsymbol{L}\Delta\varphi \qquad (2.2.3.5)$$

where

$$\boldsymbol{u}_n^E = \boldsymbol{K}^{-1}\left(\boldsymbol{p}_n + \boldsymbol{Q}_n - \boldsymbol{L}\varphi_{n-1}\right) \qquad (2.2.3.6)$$

The only unknown in *Equation (2.2.3.4)* is the change in plastic hinge rotations during this step. In order to evaluate it, we use *Equation (2.2.3.2)* and substitute $\boldsymbol{u}_n$ as follows

$$\boldsymbol{L}^T\left(\boldsymbol{u}_n^E - \boldsymbol{K}^{-1}\boldsymbol{L}\Delta\varphi\right) + \boldsymbol{H}\varphi_{n-1} + \boldsymbol{H}\Delta\varphi = -\boldsymbol{x}_n + \boldsymbol{T}_n \qquad (2.2.3.7)$$

This equation can be simplified to

$$\boldsymbol{Z}\Delta\varphi + \boldsymbol{I}\boldsymbol{x}_n = -\boldsymbol{L}^T\boldsymbol{u}_n^E - \boldsymbol{H}\varphi_{n-1} + \boldsymbol{T}_n \qquad (2.2.3.8)$$

with $\boldsymbol{Z} = \boldsymbol{H} - \boldsymbol{L}^T\boldsymbol{K}^{-1}\boldsymbol{L}$ and where all the unknowns quantities are grouped on the left–hand side of the equation.

Equations (2.2.3.4) and *(2.2.3.8)* form the basis of the numerical algorithm which is employed. Firstly we compute $\boldsymbol{Q}_n$ and $\boldsymbol{T}_n$ as described in the previous section. *Equation (2.2.3.4)* is then used to compute $\boldsymbol{u}_n^E$ and obtain an estimate for the conjugate force (moments) vector

$$\boldsymbol{x}_n^{est} = -\boldsymbol{L}^T \boldsymbol{u}_n^E - \boldsymbol{H}\varphi_{n-1} + \boldsymbol{T}_n \qquad (2.2.3.9)$$

We the start with an iterative process where we solve for $\boldsymbol{x}_n$ and $\Delta\phi_n$ by recognising that each row in the set of algebraic equations of *Equation (2.2.3.8)* represents a possible plastic hinge in the structure. The are effectively two unknowns, namely the moment and the plastic rotation at each hinge position. By making use of the fact that when the estimated moment at a possible hinge position does not exceed the yield value the incremental plastic hinge rotation is zero, and when the estimated moment does exceed the yield value both moments and incremental plastic hinge rotations are unknown, but linear related by

$$X_n = \pm M_p + s\phi_{n-1} + s\Delta\phi \qquad (2.2.3.10)$$

In the case where the component of $\Delta\phi$ is zero, the appropriate column of $\boldsymbol{Z}$ is replaced with the corresponding column from the identity matrix and the components of $\boldsymbol{x}_n$ treated as an unknown. When the component of $\Delta\phi$ is unknown, the appropriate entry in the right-hand side of the equation is adjusted by subtracting $\pm M_p + s\varphi_{n-1}$, and the $\boldsymbol{Z}$ matrix is adjusted by adding s to the corresponding diagonal term. The solution to the set of equations must provide a consistent set of results. By this we mean that where the components of $\Delta\phi$ are zero, the computed value of conjugate (moment) force components must satisfy the yield condition. If it does not, we compute a new estimated conjugate force vector

$$\boldsymbol{x}_n^{est} = -\boldsymbol{L}^T \boldsymbol{u}_n^E - \boldsymbol{H}(\varphi_{n-1} + \Delta\varphi) + \boldsymbol{T}_n \qquad (2.2.3.11)$$

and iterate until we obtain consistent results. Once we have $\Delta\phi$, we substitute into *Equation (2.2.3.4)* to compute $\boldsymbol{u}_n$.

Implicit in the formulation and algorithm described above is the possibility of plastic hinges forming in all of the elements connecting to a node. This will result in a local mechanism being formed at the node and the solution algorithm will fail. To overcome this problem, it is necessary to prevent the plastic rotation of any one of the possible hinges. This means that the increment of plastic rotation must be set to zero during the load step. It is important to choose the hinge position that will provide consistent results at the end of each step. We have based our choice of this hinge location on the ratio

$$\frac{\textit{actual moment at hinge}}{\textit{yield moment at hinge}} \qquad (2.2.3.12)$$

At nodes where the estimated moments of all possible hinge positions exceed the yield values, the hinge location for which this ratio has the lowest value is prevented from yielding by setting its plastic rotation to zero.

2.2.4 Numerical Analysis

The following examples have been selected to demonstrate the capabilities of *Quark*. The first four examples are taken from Bird (1995), and the final one from Chen(1994). The results obtained by analysing the structures in *Quark* are for examples one to four are the same as those in Bird (1995). In example 1, the load factor versus displacement curves where consistent with results obtained from STRUPL cases. The material behaviour was assumed to be elastic perfectly plastic. In the second example, detailed results where available only for the perfectly plastic case and in examples three and four only the collapse load factors assuming perfect plasticity were available for purposes of comparison. These analyses are intended only to illustrate the qualitative behaviour of the structures and thus in the third and fourth example dimensionless quantities are used. Example five takes a closer look at braced frames subject to factored static loads. The frame is loaded to its limit of resistance and the analysis is compared with results obtained using the refined plastic hinge analysis method.

2.2.4.1 Single-storey Rectangular Frame

The frame depicted in *Figure 11* consists of steel sections with cross-sectional area $A = 0.02$ m^2, second moment of area $I = 0.000067$ m^4 and Young's modulus $E = 210$ GPa. The plastic yield moment for the members is $M_p = 260$ kNm. In *Figure 12* the load factor is plotted against the horizontal displacement of the node 2 and the vertical displacement of node 3 for the perfectly plastic case. *Table 1* lists the load factors at which the hinges are formed.

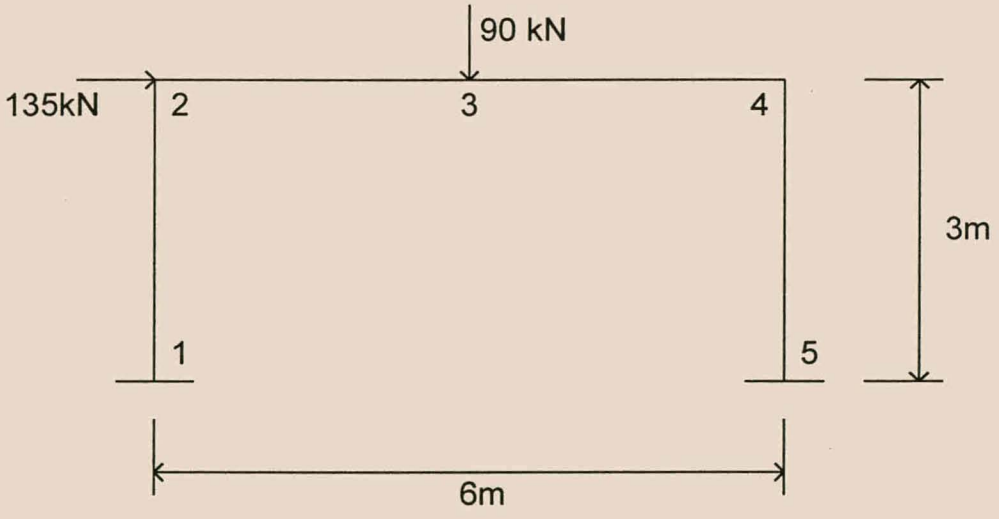

Figure 11 *Single-story rectangular frame*

Hinge	Load Factor
5	1,675
4	1,930
1	2,050
Yield	2,215

Table 1 *Results for single-storey frame*

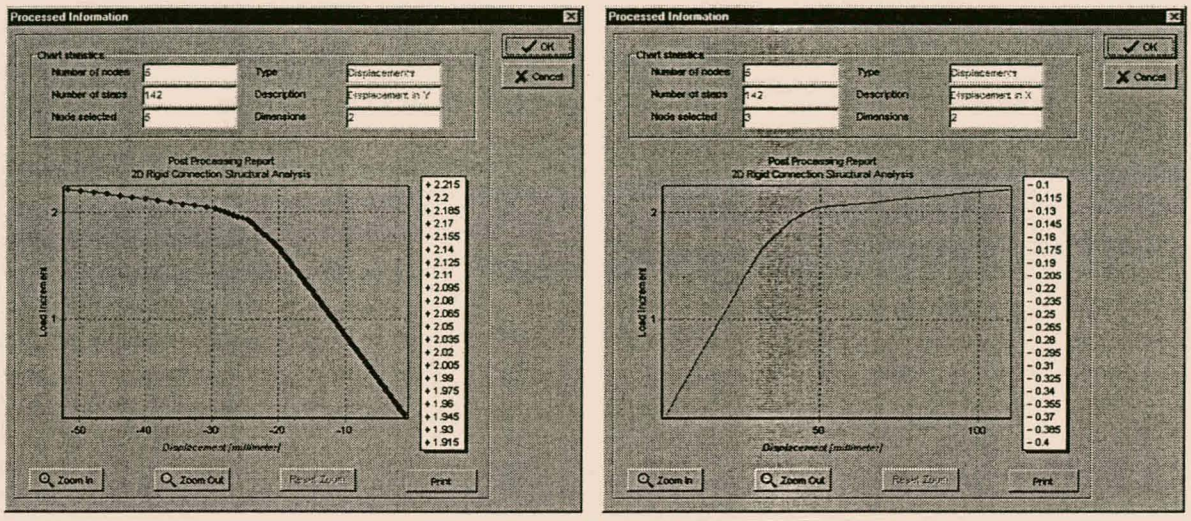

Figure 12 *Results for single-storey frame*

2.2.4.2 Two-storey Rectangular Frame

The frame analysed in this example is illustrated in *Figure 13*. The structure contains four different steel elements namely, the lower columns, the lower beams, the upper columns and the upper beam, all with Young's modulus of 200 GPa. The properties of the elements are given in *Table 2*.

Element	$I_z(m^4)$	$A_x(m^2)$	$M_p(kNm)$
Lower columns	$22{,}2 \times 10^{-6}$	$4{,}73 \times 10^{-3}$	76,95
Lower beam	$85{,}1 \times 10^{-6}$	$4{,}94 \times 10^{-3}$	151,41
Upper columns	$17{,}2 \times 10^{-6}$	$3{,}79 \times 10^{-3}$	60,56
Upper beam	$48{,}9 \times 10^{-6}$	$4{,}17 \times 10^{-3}$	105,24

Table 2 *Section properties for two storey rectangular frame*

Hinge	**Load Factor**
1	1,855
2	1,990
3	2,215
Yield	2,230

Table 3 *Results for two storey rectangular frame*

The plastic hinges, numbered in the order of formation, are also shown in *Figure 13*. *Table 3* lists the load factors at which the hinges are formed. A perfectly plastic case ($s = 0$) was investigated. It was found that the structure collapsed at the formation of the third hinge and since the structure is statically indeterminate to the sixth degree, the failure mechanism is non-regular. A regular mechanism is defined as one that forms with $n + 1$ hinges when there are n degrees of indeterminacy.

The load factor versus displacements are coincident with the results obtained with STRUPL for the perfectly plastic case.

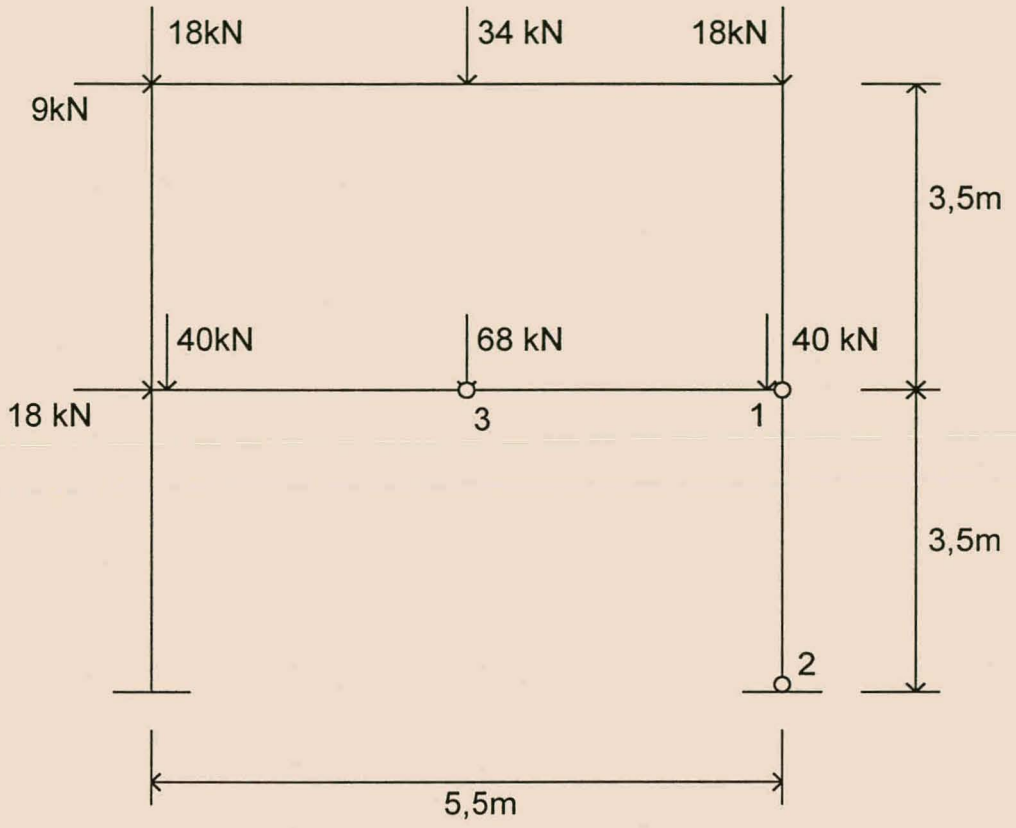

Figure 13 *Two-storey rectangular frame*

2.2.4.3 Two-bay Rectangular Frame

The structure analysed in this example is shown in *Figure 14*. The yield moment for the beams is 80, while that for the columns is 50. The load factors coinciding with the formation of each hinge are tabulated in *Table 4*. The collapse factor of 1.31 for the perfectly plastic analysis is consistent with the value obtained by Bird (1995).

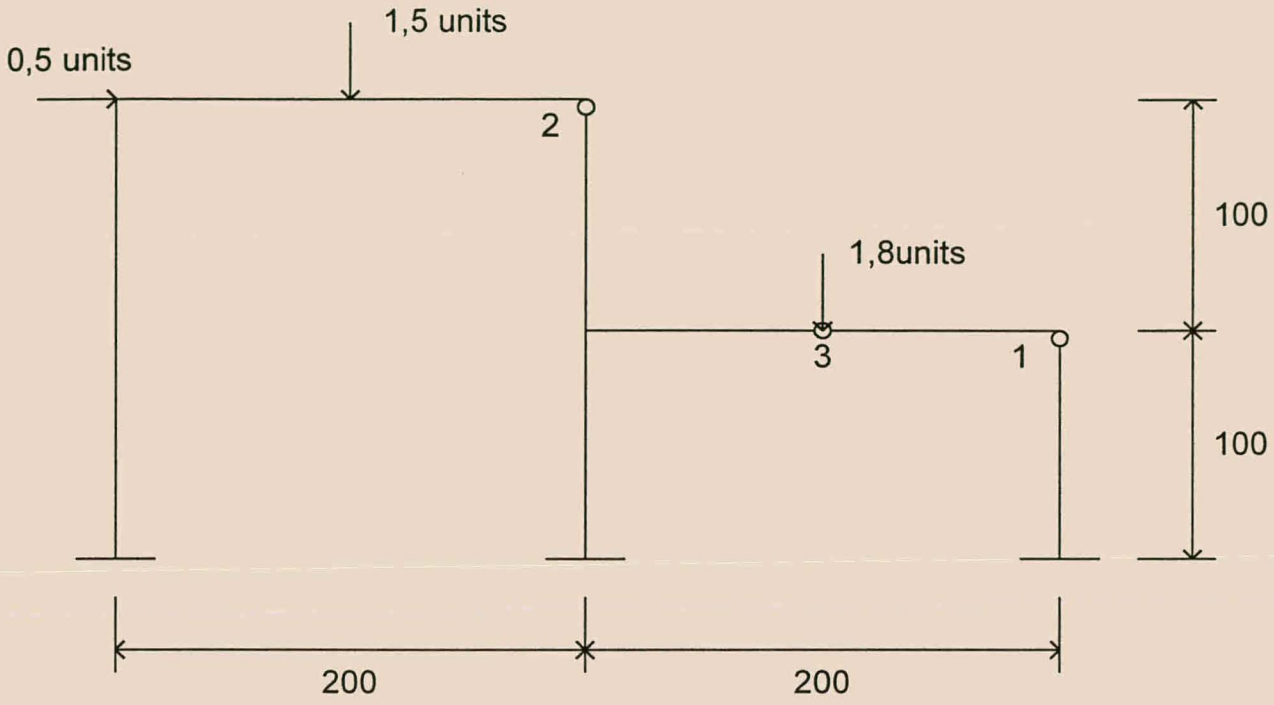

Figure 14 *Two-bay rectangular frame*

Hinge	Load Factor
1	1,12
2	1,18
3	1,23
Yield	1,31

Table 4 *Results for two bay frame*

2.2.4.4 Rectangular Frame with Distributed Load

The frame shown in *Figure 15* is subjected to a uniformly distributed vertical load of magnitude of 1 load unit per unit length, and a horizontal point load of 1 unit. The yield moment is 1 throughout. In this case the position of the maximum moment in the element with the distributed load is not know at the outset and it would be necessary to increase the number of elements to locate the position with any measure of accuracy. To model the structure correctly the element carrying the distributed load was divided into ten elements. The results for this example are tabulated in *Table 5*. The collapse factor of 2,725 for the perfectly plastic case is consistent with the value obtained by Bird (1995).

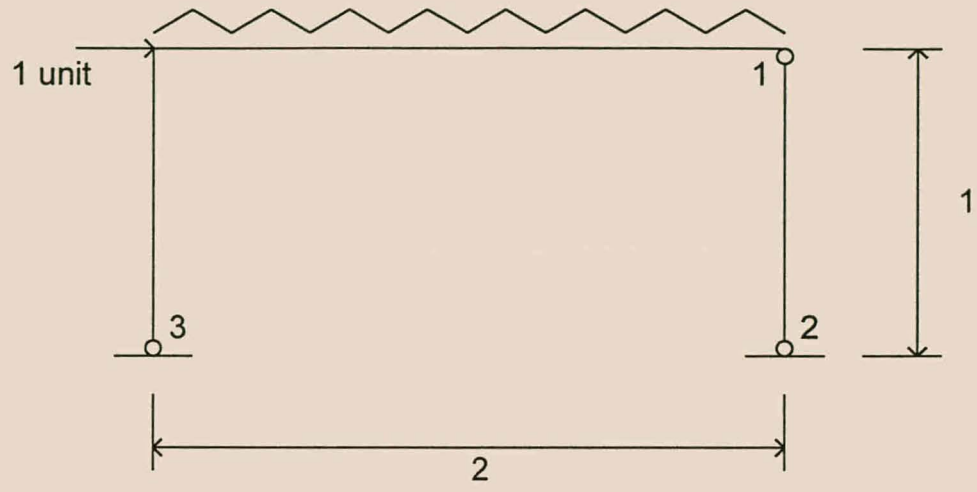

Figure 15 *Frame with distributed load*

Hinge	Load Factor
1	2,140
2	2,185
3	2,710
Yield	2,725

Table 5 *Results for frame with distributed load*

2.2.4.5 Rectangular Braced Frame

The structure analysed in this example is shown in *Figure 16.* The loads applied to the structure are as indicated in *Figure 16.* The Young's Modulus of the elements is 200 GPa and the yield stress is 250MPa. The section properties are given in *Table 6* below. The results are presented in *Table 7*. The trusses are indicated in *Figure 16* by the two triangles below the sections.

Element	$I_z(mm^4)$	$A_x(mm^2)$	$Z_p(mm^3)$
Columns	$4{,}5785 \times 10^7$	$5{,}8889 \times 10^3$	$4{,}9817 \times 10^5$
Lower beams	$2{,}1228 \times 10^8$	$6{,}6451 \times 10^3$	$1{,}0897 \times 10^6$
Upper beams	$1{,}5609 \times 10^8$	$5{,}8839 \times 10^3$	$8{,}8490 \times 10^5$
Trusses	$1{,}6316 \times 10^5$	$7{,}6774 \times 10^2$	-

Table 6 *Braced frame section properties*

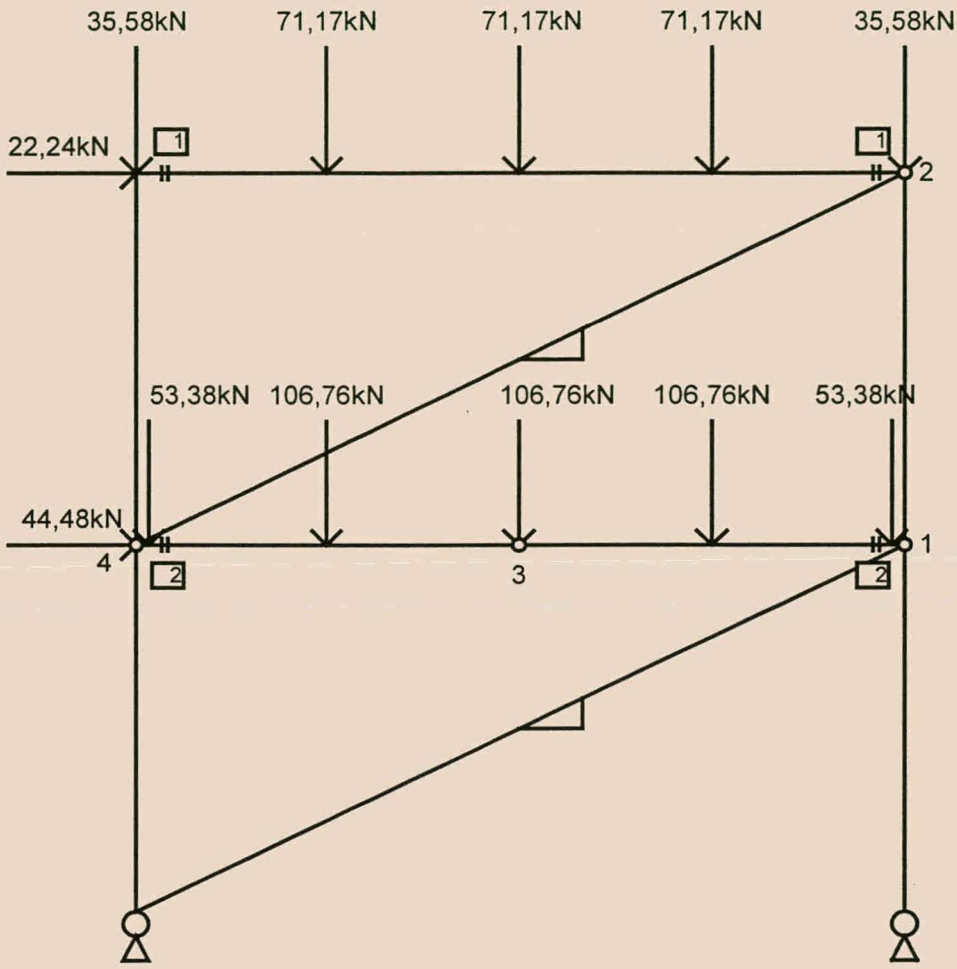

Figure 16 *Rectangular braced frame*

Hinge	Load Factor
1	1,750
2	1,810
3	2,010
4	2,020
Yield	1,230

Table 7 *Results for braced frame*

It can be seen from the results obtained from the above analysis that the results are in line with what was obtained by Chen. The reason for the lower yield factor is due to the fact that the analysis method applied here takes into consideration the second order effects of the applied loads.

If the same structure is analysed a second time with the consideration of the semi-rigid connections, the results obtained are presented in *Table 9* below. The properties of the semi-rigid connections are indicated in *Table 8.* The values of the *Ultimate Moment Capacity* of the properties used for the semi-rigid connections are different from those specified by Chen. The

values used during this analysis corresponds to the *Ultimate Moment Capacity* of the beam element in which the semi-rigid connection is located.

Property Number	Ultimate Moment Capacity (kNm)	Initial Connection Stiffness (kNm/rad)	Shape Factor (n)
1	220,00	23320,00	1,57
2	271,00	108100,00	0,80

Table 8 *Semi-rigid connection properties*

It is the opinion of the author that the results obtained by this analysis, *Table 9* below, with the above specified parameters are realistic. Because of the reduced stiffness of the complete structure, caused by the semi-rigid connections, one would expect the structure to fail at a lower yield load factor.

Hinge	Load Factor
1	1,059
2	1,076
3	1,076
Yield	1,076

Table 9 *Results for braced frame with the consideration of semi-rigid connections*

PART II: THE THEORY OF OBJECT-ORIENTED SOFTWARE

This section explains some of the concepts behind object-oriented programming techniques and principles. To achieve the final product the object-oriented programming techniques have to be combined with the structural analysis procedures.

Before this combining can take place it is important to have an appreciation for the basic principles underlying the components involved. This section steps through the object-oriented principles while with reference to structural analysis component. The chapter is concluded with a section that briefly discusses the methods that need to be applied when moving from the object-oriented analysis stage to the final stage of object-oriented programming.

CHAPTER 3

Theory, Design and Implementation of Software

3.1 Object oriented programming concepts and implementation

3.1.1 Overview

Since its inception some twenty years ago, structured programming has become a well-known technique. Structured programming however, imposes limitations on the extensibility of programs – limitations that a programming technique, known as *object-orientation*, removes or diminishes. [Reizner, Wirth]

It is common experience that programs need maintenance. If they are useful, their capabilities will expand over time. Unfortunately, even excellent structuring often fails to make the addition of features a mere local change in the program text. Instead, the places that need change are numerous and spread over many modules. Making a change in such a program is not only tedious – it is error-prone. All too easily one of the necessary changes is overlooked or one of the consequences of a change is ignored.

Clearly, a design technique that helps make feature upgrades a simpler task is of utmost value. In the world of engineering and computers there are three major design paradigms:

- Engineering Design

  In engineering design standard components are used. Each component has a well-defined external interface, e.g., a transistor has three terminals and relatively simple laws describe the relationship of voltage and current between these terminals. The designer need not concern himself with the (sometimes highly complex) inner workings of a component. The designer only concerns himself with the externally accessible properties of the component, namely the interface. An engineering design can be quickly envisaged and designed since it utilises a vast array of standard components. It is also reliable because each component is already debugged and tested.

- Procedure-Oriented Design

  For the past four decades several similar procedural design methods have been used to develop software systems. All of these paradigms are use a structured (as mentioned above), or top-down, approach to design. In structured (top-down) design, the main task is partitioned into smaller subtasks, the subtasks are then further partitioned, and so on, until the subtasks are simple enough to be programmed in some procedural language. A fundamental feature of procedural design is the separation of procedures and data. The design process is based on the subdivision of tasks, meaning procedures. The data on which they operate is secondary to the design process. Thus, procedural design is the decomposition into procedures. As a result of this focus on the procedures, the use of standard components does not fit very well into

procedural design. As a consequence, software developers have accustomed to building almost everything from scratch when developing a new application and this causes reliability to suffer and development schedules to run overtime.

- Object-Oriented Design

  Object-oriented design is the first fundamental change of paradigm in almost half a century of software development. An object is a combination of data and procedures, which operate on the data. Contrary to procedural design, the data and procedures are considered as a single entity. In object-oriented design software is developed by designing and combining software objects. Its emphasis is on reusable standard components has more in common with engineering design than it has with procedural design. Object-oriented design is a process of decomposition into objects and the object-oriented paradigm focuses on programming in the large: on how programs should be organised, on how code can be reused and extended, and how to manage the complexity.

Substantial progress in the direction of reusable and extensible programming systems is being made through object-orientation. Object-orientation techniques present a number of benefits to programming:

- The greatest problem in programming is complexity. The larger and more complex a program is, the more important it becomes to decompose it into small, comprehensible parts. Object-orientation programming provide several constructs to facilitate this process.
- From the outset objects are designed for reuse. The first object-oriented applications might be more difficult to develop than a traditionally developed application, but subsequent applications will be much easier to develop since they can reuse many of the objects which where created for other applications. Over time, a highly reliable library of reusable objects can be compiled. There are also a large number of third-party object libraries available for use.
- Object-oriented programmes can be more reliable because they use many standard tried-and-tested parts. Less new code is written for each application as more code is pulled from highly reliable libraries.
- Object-oriented programming makes it possible to develop extensible systems. Extensibility means that an existing system can be made to work with new components without have to be modified. This is one of object-oriented programming great advantages and distinguishes it from conventional programming techniques.

Object-orientation has four very specific fundamental elements. In procedural programming languages, procedures are separated from the data on which they operate (separate state and behaviour). In object-oriented programming languages the data and related procedures are fused into one entity, namely an object. The data defines the state of the object while the associated procedures define the behaviour of the object, e.g., a vector object. Each object has a certain behaviour as viewed from the outside. This refers to the term abstraction. The behavioural elements, procedures, are used to manipulate the object. An object should be able to hide some of its elements, data or procedures. This hides the detailed and sometimes fragile internal workings of an object from the software that uses it. This is called encapsulation and both this and abstraction is supported by means of access control mechanisms. The fourth and last fundamental element of the OOP (object-oriented programming) language is inheritance. Object-oriented programming provides the ability to create new objects from existing objects by a combination of adding to, and modifying an existing object's characteristics. This is accomplished by letting the

new object inherit the original object's characteristics, and then making incremental modifications and additions. An inheritance tree expresses hierarchical relationships amongst objects. Complex systems usually exhibit a hierarchical structure. Inheritance provides a means to express these hierarchical relationships in software.

3.1.2 Objects and Classes

The ability to recognise physical objects is a skill that humans learn at a very early age, e.g., a coloured ball. From this perspective of human cognition, an object is any of the following:

- a tangible and / or visible thing, e.g., a frame member, a person.
- something that may be apprehended intellectually, e.g., a vector, a matrix, and an applied point-load.
- something toward which thought or action is directed, e.g., I kick the ball; the load is applied to the structure.

For software development, real-world objects are not the only objects of interest. During the design process new abstract objects are invented to collaborate with other objects in order to achieve some desired behaviour, e.g., in some vector or matrix libraries there are indexing / subscripting helper objects for working over subscripts or ranges. From the perspective of software, an object can be defined as follows:

> An object represents an item, unit or entity, either real or abstract, with a well-defined role in the problem domain. It has state, a well-defined behaviour and a unique identity.

The state of an object represents all of the (usually static) properties of the object plus the current (usually dynamic) values of those properties, e.g., a frame member has a specific modulus of elasticity, elastic / plastic yield capacity etc. Behaviour is how an object acts and reacts, e.g., a frame member object (in two dimensions) has six degrees of freedom and from a graphical user interface perspective reacts to keyboard and mouse input. Alternatively, behaviour can be viewed as the operations that can be performed on an object, but this is a limited yet explanatory approach. The state of an object can be viewed as representing the cumulative results of its behaviour.

The identity of an object is what distinguishes it from all other objects. Identity is synonymous with existence. The unique identity of an object is preserved over the lifetime of an object, even when the state has changed.

Of course there are things that developers should keep in mind. First of all, just as a person holding a hammer tends to see everything in the world as a nail, a developer with an object-oriented mindset might begin to think of everything as an object. There are some things that are distinctly not objects. An attribute such as colour is not an object. This attribute, however, is potentially a property of another object.

3.1.2.1 Classes

The concept of a class and an object is tightly interwoven. An object cannot be considered without regard for its class. Consider a collection of support objects, e.g., a fixed support, a pinned support and a free support. Each of these objects is a concrete object that exists in time and space. All the objects in the collection share a common structure and behaviour: they constrain the structure from moving in a certain direction. The common structure and behaviour of the collection of supports defines the class of the support objects. The class is an abstraction that represents the essence of an object. A class is a blueprint that tells the compiler what the object looks like. It defines the objects structure (data representation) and its behaviour (procedures that operate on the data). An object is simply an instance of a class.

3.1.2.2 Access Controls

An object can be viewed as providing a service. Anything using the services of such an object can be called the client of that object. Some members of an object needs to be universally visible. These members form the outside view of the object. An object should be able to hide some of its members so that they are reserved for internal use only and protected from unwarranted external tampering. This hides the detailed internal workings of an object from its clients. The public members are all accessible to clients of that class. They form the public interface of that class. The private members of a class are not accessible outside the class. They form the encapsulated, or hidden, part of the class. A class can rely on its private members for consistency because their manipulation and use are completely contained within the class member functions.

3.1.2.3 Constructors and Destructors

When defining and initialising an instance of a built-in type, e.g., an integer, the compiler creates space for the variable and initialises it. In effect, the compiler constructs the variable. When a variable of a built in type goes out of scope the compiler cleans up the variable by freeing the space for that variable. In effect, the compiler destroys the variable. An object is constructed when it is created, and an object is automatically destroyed when going out of scope or existence. When an object is created, a special function, called a constructor is invoked, and when the object goes out of scope or existence, a special function, called a destructor is invoked. A constructor turns a collection of unified data values into a valid living object. A constructor initialises the member variables, allocates any additional storage, and performs any other functions that are required to convert a chunk of memory to an object. A destructor releases any resources allocated by the objects constructor. The usual resources being acquired in a constructor, and subsequently released in a destructor, is dynamically allocated memory.

3.1.2.4 Duality of the term 'Object'

The term 'object' plays two roles in object-oriented programming

- The first role is an organising principle. The object concept is central to object-oriented programming.
- The second role refers to an instance of a class, a region of storage that contains one copy of the item specified by the class blueprint.

The major goal of object-oriented programming is to make working with software components as easy and productive as an engineer's work with standard components. In the physical and software world, an easy to use, reliable and powerful parts collection can give a designer the ability to be more productive. Instead of building from scratch, the designer can take advantage of the work of others, assembling ready-made components. Reusable software blocks are not the most profound or innovative use of object. The more significant use of an object is a foundation from which more specialised objects can be derived. In object-orientation the specialisation of a class is accomplished by means of inheritance.

3.1.3 Abstraction and Encapsulation

3.1.3.1 The Meaning of Abstraction

Abstraction is one of the fundamental ways for humans to cope with complexity. Shaw [6] defines an abstraction as follows:

> An abstraction is a simplified description, or specification, of a system that emphasises some of the system's details or properties while supporting others. A good abstraction is one that emphasises details that are significant to the reader or user and suppresses details that are, at least for the moment, immaterial or diversionary.

Booch defines abstraction as follows:

> An abstraction denotes the essential characteristics of an object that distinguishes it from all other objects and thus provides crisply defined conceptual boundaries, related to the perspective of the viewer.

Abstraction can also, in simplified terms, be viewed as the process of making something complex appear simple by focusing on its essential behaviour, and ignoring details that are immaterial relative to the perspective of the user. An abstraction focuses on the outside view of an object, and so serves to separate an object's essential behaviour from its implementation. This behaviour/implementation division is called the abstraction barrier. A client is any object that uses the resource of another object (known as the server). The behaviour of an object can be characterised by considering the services that it provides to other objects, as well as the operations it might perform on other objects. This might also be viewed as a contract between the object and all its clients.

3.1.3.2 The Meaning of Encapsulation

The abstraction of an object should precede the decisions about the implementation. Once an implementation is selected, it should be treated as a secret of the abstraction and hidden from most clients.

Encapsulation is achieved through information hiding, which is the process of hiding all the secrets of an object that do not contribute to its essential characteristics. Abstraction and encapsulation are complementary concepts: abstraction focuses on the observable behaviour of an object, whereas encapsulation focuses on the implementation that gives rise to this behaviour. It is generally accepted that for abstraction to work, implementations must be encapsulated. In practice this means that each class must have two parts: an interface and an implementation. The interface of the class captures the outside view, encompassing the abstraction of the behaviour common to all instances of the class. The implementation of a class consists of the representation of the abstraction, as well as the mechanisms that achieve the desired behaviour. Abstraction helps people to think about what they are doing, while encapsulation allows program changes to be made reliably with limited effort.

In summary, Booch defines encapsulation as follows:

> Encapsulation is the process of compartmentalising the elements of an abstraction that constitute its structure and behaviour; encapsulation serves to separate the contractual interface of an abstraction and its implementation

Intelligent encapsulation localises design decisions that are likely to change in the future. As a system evolves, its developers might find during actual use that certain operations take longer than acceptable, or that some objects consume more memory space than is available. In such situations the implementation of an object is often changed that more efficient algorithms can be applied, or so that space can be optimised by calculating rather than storing certain data. In summary the ability to change the implementation of an abstraction without disturbing any of its clients is the essential benefit of encapsulation.

3.1.4 Hierarchy

3.1.4.1 Hierarchy and Complexity

Complexity frequently takes the form of a hierarchy, whereby a complex system is composed of interrelated subsystems that have, in turn, their own subsystems, and so on, until some lowest level of elementary components is reached. The fact that many complex systems have a hierarchical structure is a major facilitating factor that enables us to understand, describe and visualise complex systems and their parts. Complex systems usually have different types of hierarchies present in the same complex system. A framework, for example, may be studied by decomposing it into frame elements, truss elements, nodes, applied loads etc. This decomposition represents a structural, or *'part of'* hierarchy. Alternatively, we can cut across the system in an entirely orthogonal way. For example, a frame element is a specific kind of element. This hierarchy represents an *'is a'*, or *'kind of'*, hierarchy.

Object-oriented programming provides constructs to express both types hierarchies. Humans normally abstract relationships between physical objects on two dimensions: *kind-of* and *part-of* relationships. Object-oriented programming thus models more closely the way we think about the world.

3.1.4.2 Hierarchy: Inheritance

In the above example a frame member was used. A frame member thus *inherits* all of the properties of a general member, and then adds some new properties and possibly redefines some of the inherited properties. Inheritance expresses the *is-a* relationship. It defines the relationship where one object shares the structure and behaviour of another object. Inheritance relationships can be expressed diagrammatically with an inheritance tree, as shown in *Figure 17*.

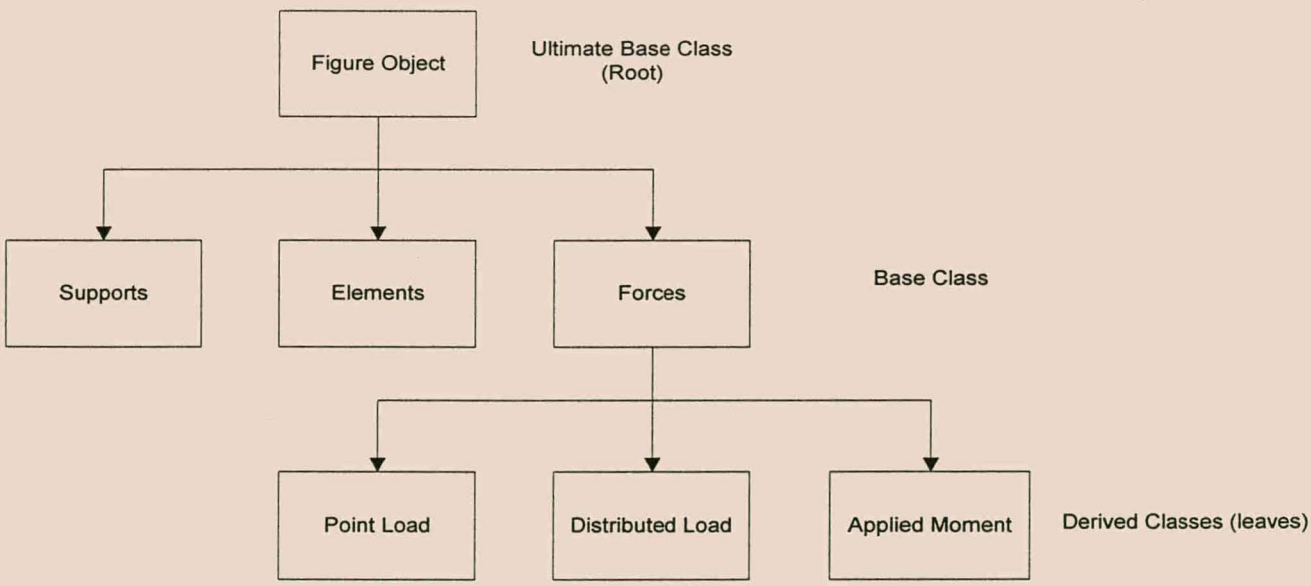

Figure 17 *A figure primitive inheritance class hierarchy.*

To illustrate inheritance in object oriented programming, consider the example of *Quark* that requires a graphical editor. The editor has to work with geometrical figures such as lines, nodes, members etc.

Typical abstractions in such a problem domain are that of different kinds of figure objects, e.g., support objects, node objects, member objects.

An abstraction for a support object might be as follows:

- A support has a colour.
- A support can be drawn and erased on the screen.
- A support can be moved, rotated, mirrored, etc.
- A support has a size, constraints, prescribed displacements etc.

An abstraction for a node object might be as follows:

- A node has a colour.
- A node can be drawn and erased on the screen.
- A node can be moved, rotated, mirrored, etc.
- A node has a centre, a size, constraints etc.

An abstraction for a member object might be as follows:

- A member has a colour.
- A member can be drawn and erased on the screen.
- A member can be moved, rotated, mirrored, etc.
- A member has a length, connected to nodes or supports, a size, etc.

We can factor out the commonalties of all the figure objects and define a new object, say a figure object, with the following abstractions;

- A figure has a colour.
- A figure can be drawn and erased on the screen.
- A figure can be moved, rotated, mirrored, etc.

A figure object abstracts the essentials of what is to be a figure. An object, such as a support object, is a specialised figure object. Similarly, a node object, is a specialised figure object. These specialised objects inherit behaviour and properties from the figure object; they may modify figure's behaviour and add new behaviours and properties. Note that the support, node and member objects have an *is-a* relationship with the figure object and thus represents an *is-a* hierarchy.

We can define classes *Figure, Support, Node* and *Member* to capture the above abstractions. If we want to utilise the *is-a* relationships, the *Support, Node* and *Member* classes should inherit the structure and behaviour of the *Figure* class. The *Support, Node* and *Member* classes should also be able to redefine the inherited *Figure* methods and add new methods.

Object-oriented programming languages provide constructs and mechanisms whereby the *is-a* relationship can be expressed, and whereby methods can be inherited, added and redefined. A class (the subclass) can be derived from an existing class (the parent class). The subclass inherits the structure and behaviour of the parent class, and may even add and redefine some of its members.

In OOP terminology, the relationship of the *Member* class to the *Figure* class can be expressed in several ways:

- *Member* is a kind of *Figure*
- *Member* is derived from *Figure*
- *Member* is a specialised *Figure*
- *Member* is a subclass *Figure*
- *Figure* is a parent/ancestor of *Member*
- *Figure* is the base class of *Member*
- *Figure* is the superclass *Member*

The classes *Support* and *Node* can be declared similarly. They will each override the inherited *draw()* and *move()* methods, and also add new support-specific and node-specific methods, respectively. The *Support, Node*, and *Member* objects reuse the inherited code of the *Figure* class. This results in less and more reliable code.

When a class is derived from an existing class, the following changes can be made in the derived class:

- New member variables can be added.
- New member functions can be added.
- Inherited member functions can be redefined.

Inheritance, or derivation, is also known as *type extension*. The inheritance mechanism does not allow the deletion of member variables and member functions. This is because a derived class is viewed as an extension of its base class. *A derived class can be used anywhere where its base class can be used.* If some members of the base class are missing, this will not be possible. Class derivation is an act of specialisation, of refinement. The tendency should be toward more and finer, not fewer and coarser.

Inheritance thus implies a generalisation / specialisation hierarchy: the subclass specialises the more general structure and behaviour of its superclass. This leads to the litmus test for inheritance: if *B* "is not a" kind of *A*, the *B* should not inherit from *A*.

3.1.4.3 Compatibility of a Base Class and Its Extensions

From the above it might look like inheritance is only a way to reuse code and thus reduce writing effort. Consider having a class, *AltMember,* in which the methods are explicitly redeclared instead of being inherited from *Figure.* There is a very important difference between *Member* and *AltMember*: *Member* is compatible with *Figure* since it is an extension of *Figure,* while *AltMember* is not compatible with *Figure* although it contains the same methods as *Member. AltMember* is a completely different type. In other words, every *Member* object is also a *Figure* object but an *AltFigure* object is not a *Figure* object.

This is a revolutionary aspect of OOP. It means that all code that work with *Figure* objects can also work with *Member* objects. This is particularly useful in assignments, when an object is assigned to a reference variable. Note that no data is lost when, for example, a *Member* is assigned to *Figure* reference. The *Figure* reference only has access to the *Figure* methods of the *Member* object and is 'blind' to the *Member*-specific methods.

3.1.4.4 Dynamic Binding

In OOP languages, pointer and reference variables have a dynamic type in addition to their static type. Considering our examples, the static type of pointer- and reference variables is the type *Figure* and the dynamic (actual) types of the variables can be either *Node* or *Member*.

The compiler does not know at compile time what the dynamic types of the function arguments for the variables (either pointer or reference) will be. It must therefore determine at run time what the dynamic type of the arguments is and then invoke the correct method. This run time mechanism is called *dynamic binding or late binding.*

Dynamic binding contrasts with *static binding,* or *early binding*, which happens with conventional function calls. With static binding the compiler knows the address of the function and generates a direct call. With dynamic binding the compiler does not know the address of the corresponding method. The address must be determined at run time using the dynamic type of the receiver object. Message sending via dynamic binding thus incurs an overhead. Dynamic binding is also known as polymorphism, meaning to have many forms. Most modern OOP languages, e.g., C++ enables dynamic binding only for methods that have been declared as *virtual.* A non-virtual method uses only static binding. A virtual method is also known as a *polymorphic method.*

Consider an example where the draw function for an array of figures is to be invoked. The array of figures can contain different kind of figure objects simultaneously, and is thus *heterogeneous.* The dynamic type of the objects in the array of figures is unknown and irrelevant at compile time. If the draw operation is to be applied to one of these variables, the programmer does not have to care about the actual dynamic type. The programmer can send a message to the variable and lets the object interpret it. The message tells *what* is to be done, the object itself determines *how* it is done. This allows one to add a new class, for example, a *Spring* class, to the application. The *draw()* method, and all other code written to work with the *Figure* class, will work with this new object without having to be recompiled. The application is thus *extensible*, meaning that existing code is not affected by the addition.

3.1.4.5 Abstract Classes

In the original definition of the *Figure* class, the implementation of the *draw()* and *move()* methods were empty, that is, they did nothing. The derived classes then provided their own implementation of these methods. It is clear that a *Figure* object cannot possibly know how to be drawn or moved. Any class derived from *Figure* therefore has to provide its own implementations of these functions. While deriving a new class from *Figure* one might forget, for example, to provide an implementation for the *move()* method. This mistake can only be determined at run time since the compiler will not generate any errors. A language like C++ provides a means to *force* a derived class to provide implementations for certain methods. The methods forced are equated to 0 and are called *pure virtual methods*, or *abstract methods*. A class containing any abstract methods is called an *abstract class*. Delphi defines methods as abstract with the *abstract* compiler directive defined after the method definition.

No instances can be made of an abstract class. An abstract base class is designed solely serve as a base class for other classes. The major role of an abstract base class is to create an interface for a family of classes with common behaviours.

3.1.4.6 Multiple Inheritance

In ordinary inheritance, also called *single inheritance*, each derived class has a single parent class. In the derived class, methods can be added and redefined in order to create a specialised version of the base class.

Figure 17 shows a class hierarchy consisting of single inheritance relationships. It has a tree structure where each base class is a single node. A class without derived classes is a leaf node. A leaf node has at most a single parent node, but it can have several ancestors. The ultimate base class is called the root class.

In *multiple inheritance* a derived class can have more than one parent class. It enables a class to inherit features from more than one base class. *Figure 18* shows an example of multiple inheritance as used in the *iostream* class library. It is an alternative to the standard C I/O library. The *iostream* class is derived from both the *istream* and *ostream* classes.

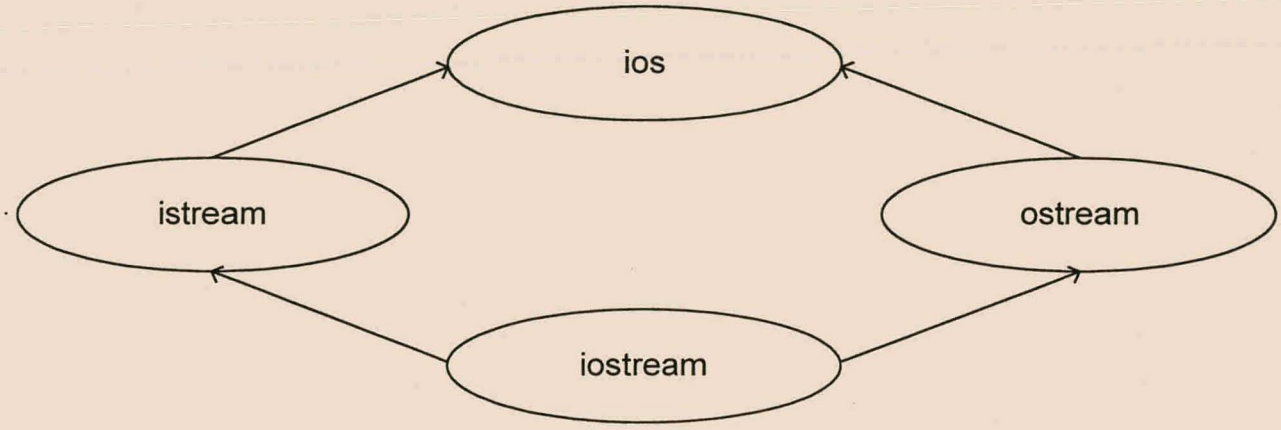

Figure 18 *Multiple inheritance in the iostream library*

Multiple inheritance does not change the nature of the *is-a* relationship. The derived class has an *is-a* relationship with each of its base classes, e.g., an *iostream* is an *istream* and an *iostream* is an *ostream.* The root class in the *iostream* hierarchy is the *ios* class. The *istream* class contains numerous facilities related to input from streams, including input conversion functions and some buffered input functions. The *ostream* class provides output conversion functions and some buffered out functions. In some applications, both input and output is required on a single file. The *iostream* class fulfils this requirement. It is both an *istream* and an *ostream*. The reason for this separation is improved safety checking of code. All that can be done with an *istream* object is input, no output operations are available. This is much safer than I/0 in C, which lets you open a file for input and then allows you to write.

Multiple inheritance is still a controversial subject. Many things that can be done with multiple inheritance can also be done (even better) with single inheritance methods. There are, however, cases where multiple inheritance can be very useful.

3.1.4.7 Hierarchy: Aggregation

It has been shown that complex systems usually have both an *is-a* hierarchy and a *part-of* hierarchy. Inheritance is used to express the *is-a* or *kind-of* relationship. It is not useful for expressing *has-a* or *part-of* relationships. Trying to express the latter relationships using inheritance is a common mistake among novices. Using objects within an object is called *aggregation, composition* or *containment*. The *part-of* relationship is thus expresses using *aggregation.*

Aggregation has been a first class part of abstract date typing techniques in languages such as Modula-2, ADA, and C. It is important to view a complex system from both perspectives, studying its *is-a* hierarchy as well as its *part-of* hierarchy. These different hierarchies are called the *class structure* and the *object structure* respectively.

3.1.5 Object-Oriented Analysis and Design

Analysing and designing of the object-oriented system is the greatest challenge faced when attempting to implement object-oriented principles. In many cases it is necessary to go through the analysis and design process more than once. During the design process the designer might be faced with a new problem. This problem, if sufficiently complex, may require a fresh application of the analysis and design steps.

3.1.5.1 The Meaning of Object-Orientation

The phrase *object-oriented* has become a marketing buzzword. Everything is claimed to be object-oriented. One might ask what object-oriented really means. The concept of an object is central to anything object-oriented. The basic idea is that an object serves to unify the ideas of algorithmic and data abstraction.

3.1.5.2 Object-Oriented Analysis

Object-oriented analysis (OOA) is a method of analysis that views the world from an object-oriented perspective. It is based on the view that objects, their relationships, behaviours, and their interactions can represent real-world problems.

During OOA the following questions are addressed:

- What are the objects in the problem domain?
- What is the desired behaviour of the system?
- What are the roles and the responsibilities of the objects that carry out this behaviour?

The result of the OOA step is the following:

- The objects that model the problem domain.
- The behaviours of these objects.
- A classification of these objects in terms of their common roles and responsibilities.

The result is a requirement specification that is independent of any programming language. It specifies (from an object-oriented perspective) what is required form the software system, i.e., *what* it should be able to do, but not *how* it should be done.

Booch defines OOA as follows:

> Object-oriented analysis is a method of analysis that examines requirements from the perspective of the classes and the objects found in the vocabulary of the problem domain.

OOA emphasises the building of object-oriented models that mirror the real-world problem domains.

3.1.5.3 Object-Oriented Design

Object-oriented design (OOD) takes the requirements specification generated by the OOA step and generates an object-oriented design. The result of OOA specifies *what* the system should be able to do. The result of OOD specifies *how* is should be done by using object-oriented methods. Another viewpoint is that the analysis step defines the problem formally and the design step creates a solution to the problem. OOD will identify many new classes and objects that are required in order to create a solution to the problem. There are also many standard object-oriented solution to certain problems (design patterns).

A programmer takes the requirements specification and creates a design by using experience in OOD, knowledge about design patterns and a lot of innovative thinking.

During OOD the following questions are addressed:

- What classes are required and how these classes are related (class structure).
- What objects are required and how they co-operate (object structure).
- Where each class and object should be declared (physical model).
- What actions change an object's state, and what actions result from a state change (dynamic model).

The result of the OOD step is an object-oriented decomposition, which consists of the following:

- The class and object structure (logical model).
- The allocation of classes and objects to program modules in the physical design of the system (physical model).
- A description showing what events causes objects to change state, and what action results from the state changes (dynamic model).

Booch defines OOD as follows:

> Object-oriented design is a method of design encompassing the process of object-oriented decomposition and a notation for depicting both logical and physical, as well as static dynamic models of the system under design.

There are two important parts to this definition:

1. OOD leads to an object-oriented decomposition.
2. Uses different notations to express the logical and physical design of a system, in addition to the static and dynamic aspects of the system.

The support for object-oriented decomposition is what separates OOD from structured (procedure-oriented) design. OOD uses class and object abstractions to logically structure systems, while structured design uses algorithmic abstractions.

3.1.5.4 Object-Oriented Programming

Object-oriented programming (OOP) takes the resulting object-oriented decomposition from the OOD step, and implements the design in a programming language. It is a mapping from the design to the programming language.

Booch defines OOP as follows:

> Object-oriented programming is a method of implementation in which programs are organised as co-operative collections of objects, each of which represents an instance of

some class, and whose classes are all members of a hierarchy of classes via inheritance relationships.

There are three important parts to this definition:

1. OOP uses *objects*, not algorithms, as its fundamental building blocks (the *part-of* hierarchy).
2. Each object is an *instance* of some class.
3. Classes are related to each other via *inheritance* relationships (the *is-a* hierarchy).

A program may appear to be object-oriented, but if any one of these elements is missing it is not an object-oriented program. Programming without inheritance is distinctly not object-oriented; it is called *programming with abstract data types*.

Note the OOP uses the available mechanisms in a particular language that support classes, objects and inheritance in order to implement an object-oriented program. The mechanisms may differ between different languages, but the program remains object-oriented. By this definition some languages are object-oriented and some are not. For a language to support inheritance it must be able to express the *is-a* relationship among types. If a language does not support inheritance it is not object-oriented. If a language supports objects but not inheritance it is called *object-based.* By the above definition languages such as Smalltalk, C++, Eiffel, Objective-C, Delphi, Java and CLOS are all object-oriented, while the language ADA is object-based.

3.1.5.5 The Available Mechanisms

OOP consists of the mapping of the results of the design step into an object-oriented programming language, which in the case of *Quark* is Delphi. The following mechanisms are instrumental in doing so:

- **Abstract Data Typing**: This mechanism allows the creation of user-defined types together with a full set of operations for each type. Internal details are hidden through encapsulation, while abstraction embodies the concept the date type represents.
- **Inheritance and Dynamic Binding**: This mechanism allows the creation of a specialised version of a class from an existing class. Common behaviours are made explicit by using inheritance. Inheritance expresses the *is-a* relationship. Dynamic binding allows the writing of extensible code.
- **Aggregation**: This mechanism allows the creation of classes, which contain objects (instances) of other classes. It expresses the *part-of* relationship.

3.1.5.6 Object Behaviour Analysis

The analysis method described here is known as *Object Behaviour Analysis*:

1. **Define the problem**: The first step in object behaviour analysis is to describe, in reasonably precise terms, what the problem is. The description should use terms that are natural to the application area. For example, the description of a structural analysis system would use words such as structure, loads and displacements.
2. **Develop an informal strategy**: The next step is to write an informal prose description of a method that will solve the problem. The description, like the definition of the problem, uses terms for the application area. The result of this step is a set of scripts which describe the sequence of actions that must be taken to solve the problem.

3. **Formalise the strategy**: This step has several parts:
    - Examine the description of the informal strategy developed in step 2, listing the objects mentioned in the strategy. They are easy to spot because English nouns represent them.
    - Examine the description again, this time looking for verbs that describe operations that are performed on the objects. The operations define the behaviour of the objects.
    - Group the objects identified in step 3 bullet one together with the operation to which they apply. The outcome of this step is a list of objects an their respective behaviours (operations). For a structural analysis system, the operations on the structure might include adding nodes and members etc.
4. **Classify the objects**: The next step is to group the objects according to some similarity of *behaviour*. The commonality of these can be abstracted into an abstract object. For example A *Node* can be drawn and moved, a *Member* can be drawn and moved, a *Support* can be drawn and moved. The common behaviours, drawing and moving, can be abstracted into an abstract. *Figure* object. The outcome of this step is a diagram of initial hierarchical relationships between objects, based on forming abstract objects using the criterion of similar behaviour.
5. **Identify relationships**: Working with the outcomes of the previous steps – objects, their behaviours and classification according to behaviour – this step involves drawing a *preliminary* sketch of the objects in relation to one another. The relationships are usually the following :
    - The *is-a* relationship, which indicates a natural hierarchical grouping of objects. For example, *Fixed Supports, Free Supports* and *Pinned Supports* all have similar behaviours which were abstracted into shape class in step 4. In this step the following additional natural relationships can be identified: a *Fixed Support* is a *Support* and a *Support* is a *Figure*. The outcome of this step is a finer hierarchical grouping of objects.
    - The *part-of* or *has-a*, relationship. This relationship identifies objects, which are contained within other objects. For example, a *Structure* has *Nodes* and a *Structure* has *Supports*.
    - Communication is another form of relationship among objects. One object may request information (or some behaviour) from another object, send information to it, refer to it, or know about it. For example, an *Applied Point Load* object will notify the *Structure* object if there is no *Node* object to connect to.
6. **Model the process**: Finally, it is important to determine which objects initiate activities, and identify the sequence of activities. The scripts developed in step 2 to 'run' the model of the application must be used. That is, given the objects, behaviours, and relationships defined thus far, can the scripts are enacted using the object behaviours that have been defined? The lifetime of objects, as well as their status at different parts of the cycle, must also be specified.

PART III: THE COMBINATION OF THE THEORIES

To complete the study, the theories developed and methods studied have to be combined. This is achieved by examining the principles and results of the software developed. The focus is not on the mathematical algorithms, but rather on the features of *Quark*.

The chapter is broken up in two main themes namely 'what has been implemented' and 'what must be implemented' to achieve the ultimate goal and finally it closes off with the overall conclusions for the thesis.

CHAPTER 4

The Design and Coding of *Quark*[†]

4.1 Overview

The main aim of this thesis was twofold: to extend the algorithms for the plastic analysis of rigidly connected plane frames and the implementation of existing and extended algorithms into a single software system that could deal with these complex numeric analysis algorithms.

This topic posed a number of unique challenges. Firstly, the combination of the thesis requirements namely analysis, design, development, implementation and testing demanded a high level of expertise in two distinctly different disciplines. Secondly, it was important to maintain a balance between the two disciplines in order not to deviate from the topic of the thesis. Thirdly, when developing a solution in one of the disciplines an equally elegant solution had to be found for the other discipline to ensure that the two were always compatible.

Quark was designed specifically for the plastic analysis of *Plane Frames.* All the drawing primitives designed for the software was aimed at facilitating a user to quickly and effectively design the structures. The procedures and general use of the software was designed with the same idea in mind. The numerical analysis of plastic structures is a complex and computer resource intensive operation. Special numerical structures were designed and implemented to allow users to create structures of different sizes to analyse, a limitation occasionally imposed on designers using earlier and more primitive software systems.

To ensure that a user could quickly and correctly interpret the results produced by *Quark,* a post-processor or data display functionality was added. Users these days expect software to be friendly and functional and to facilitate certain tasks, one of which would be to interpret vast amounts of numerical data.

[†] The existence of particles with charge smaller than the charge carried by the electron and proton was proposed in 1963 by Murray Gell-Mann and George Zweig. They called these particles *quarks*. For more than thirty years the proof of existence of quarks eluded scientists. When considering the task at hand to complete the software and the thesis, the author shared the same sentiment as all of those scientists looking for something but just never quite reaching that point of conclusion. That is the reason for naming the software *Quark.*

4.2 Graphical Interface and Pre-Processor

One of the greatest challenges in designing and coding *Quark* was to simplify the input of numerical data for analysis. Many a Structural Engineer, especially those with a number of years experience, have been through the painful process of entering thousands of co-ordinates, links, forces, etc. into a text file and then using that text file and some DOS based FORTRAN program to analyse the data. Adding to the frustration, there would often be a message that somewhere in the text file a comma or full stop was missing and that without that vital piece of information the system would not be able to complete the task at hand.

Another limitation of previous systems, especially the earliest ones, was that the data structures implemented by them where static. The size of internal data structures, e.g., the matrices and vectors were fixed resulting in tremendous practical limitations which rendered the software almost useless for everyday applications.

The previous version of the academic software used to test some of the advanced analysis algorithms, had these limitations. In order to overcome these problems a graphical interface, also called the pre-processor, was designed and coded. The idea was to focus its functionality to the use and analysis of the steel plane frames. After the initial design the functionality was extended to a three dimensional environment to facilitate later extensions of the algorithms to three dimensions.

Users have free movement within the three dimensional environment. They can translate, rotate and zoom in this model and always have the ability to manipulate, adjust or add to the model. Visualisation is the key to better understanding structures and the ability to manipulate that visual interface or the model itself gives the user the best possible chance to fully make use of the potential of *Quark*. The system provides a user with three sub-view's of the structure in various orientations for a better interpretation of the model when manipulating the model in the three dimensional space.

The interface was designed in a now industry standard Windows style that allows the user to 'pick-and-click' to put the structure together. People do make mistakes and the interface had to provide methods to correct and adjust structures after drawing or 'building' them. The interface therefore provides users with the standard methods used in all industry systems, i.e., cut, paste, click-and-drag etc. This allows a user to adjust quickly and visually, models to suit them. Because it is difficult to design a large three dimensional model in three dimensional space when you are using a two dimensional computer screen, *Quark* allows users to design, build and adjust models with the aid of input text grids. The grids extend to allow the user to manipulate most of the parameters of the objects in the model. To manipulate the model in this way the model can either be opened in the form of a drawing or in an Explorer style. The manipulation of the interface is extended to include set up or control parameters. In other words the user can manipulate the environment that the model is built in and if the user requests, multiple models can be built or analysed at the same time by opening more that one model in a *Multiple Document Interface* (MDI) style.

The pre-processor contains various primitives specifically suited to building structural models. These primitives included nodes, different supports etc. They all have predefined behaviours and methods that apply to them that allow the objects to interact with the user and with each other and to behave like the 'real-life' objects they are modelled on. Some rules applying specifically to

structures and the objects in structures have been included in the software. In other words a user would not be able to for example create a frame or truss member without having at least two nodes, two supports, or a node and support to connect the ends to. By the same token, the system will not allow a user to connect both ends of a member to one support or one node. The reason for implementing these rules was to avoid any obvious problems in the structure during analysis. The system warns the user of all possible problems with the model before continuing to the analysis section of the software. In extreme cases the software will not allow the user to continue until the error has been resolved.

Methods and operations specifically for the purpose of the software were also implemented. The system, for instance, allows users to dynamically add nodes to already existing members. This particularly useful when the members are too long (slender) and the plastic analysis routines can not converge successfully. The members in such cases must be 'shortened' to avoid the problem.

It was desirable to design the interface in such a way as to always have all information in the model accessible and to provide easy access to the information.

4.3 Dynamic Numerical Structures and Analysis Algorithms

One of the major limitations of the previous system that had to be overcome was the move from a fixed size model to a dynamic sized model. In other words a user should have the ability to build a model with four nodes and a model with a thousand nodes.

Being a structural analysis software system implied that matrices and vector structures would be used extensively in the software and that defining these data structures statically would remove any possibility of achieving a dynamic system able to satisfy the criteria mentioned above. The requirement was achieved by defining and building a dynamic data structure class inside of the software with the aid of pointers. This base class was extended to include the vector class, matrix class and a three dimensional matrix class. These dynamic objects allow *Quark* to achieve its flexibility.

To ensure the proper working of these objects the system monitors the creation or destruction of the objects. This has been extended to a data / processor output interface for the user to manually monitor possible memory leaks in the system. This interface includes the reporting on other dynamic structures in the system like the lists used to model and maintain the objects in the structure and track the threads used to manipulate the three sub-views used. Because memory is such a critical issue in computers doing large volumes of numerical calculations, the interface tracks and reports on the use of memory in the system. The information provided is an extension of information provided by the operating system to allow the user a better understanding of the use of memory in the system.

The primary objective of the system is to correctly calculate or analyse the models designed and build by the end user. To ensure this, a number of procedures or methods, rules and interfaces where implemented in *Quark*. As mentioned above, *Quark* has built in rules that verify the correctness of a model before allowing the user to analyse such a model. These rules are extended to thoroughly scrutinise the model before entering an analysis routine inside the analysis interface.

Quark gives a user the ability to analyse a structure with three different algorithms. They are an elastic three dimensional analysis routine for structures containing frame and truss elements, a two dimensional plastic analysis with semi-rigid connections for structures containing frame and truss elements and a two dimensional plastic analysis with rigid connections for structures containing frame and truss elements. Because of the complexity of the calculations and the duration or time used to perform these calculations the analysis interface allows the user to specify certain analysis criteria. These criteria involve alternative matrix analysis methods and recovery algorithms in the case of the calculation procedure failing. To ensure that the user is informed of all processing transactions the analysis interface gives the user the opportunity to employ this option.

The three dimensional analysis algorithm was included in *Quark* to ensure that the system could be used for other applications than just the plastic analysis of plane frames.

4.4 Post-Processor

Interpreting the output from a numerical analysis can be a complex task without any visual aids. To ensure that users can quickly and accurately interpret the output of the analysis produced by *Quark* the post-processor interface was designed and implemented.

The interface has three main output possibilities. Firstly, to display the output in the form of a displaced structure in the pre-processor interface. This allows the user to quickly interpret the values produced by the calculations. Additional information is available to the user at this point in the form of screens presenting the calculated output. Secondly, the output from the calculations is produced in the form of tabulated numerical values (grids). These grids enable the user to inspect the precise values represented in the graphical format. The data output in these grids is formatted in a report style and can be printed directly from *Quark*. Thirdly a chart interface allows the user to plot any of the various types of output data produces by *Quark*. Plots of displacements, nodal reactions or nodal moments, either elastic or plastic, are available.

The post-processor lastly contains an algorithm to print out the model built inside the pre-processor of *Quark* as well as the displaced structure.

4.5 Future developments

There are many areas in which the functionality of *Quark* can be extended to ensure more general use of the software. The various analysis capabilities implemented in *Quark* can be extended to ensure better performance from the software.

4.5.1 The Graphical Interface

The graphical interface of the pre-processor can be extended by:

- Adding a more general set of primitives to the drawing primitives currently available, e.g., to be able to either draw more general models with lines etc. without affecting the model as well as more specific structural primitives like spring support objects. The addition of structural support primitives will have a direct impact on the analysis interface inside *Quark*.
- Adding more generalised drawing operations and figure manipulation processes and methods. This would include operations to manipulate non-structural primitives, e.g., connecting a line object to a ellipse or, grouping figures together to form larger objects and to more interactively be able to manipulate structural primitives, e.g., to drag a spring support object and have the governing parameters displayed dynamically on the screen.
- The addition of viewing manipulation procedures to include the ability to change the angles at which the three sub-views are displayed and to extend the display functionality from orthographic only to include oblique and perspective viewing.
- Giving the current primitives a more 'real-life' look. For example, a member can be changed from a line drawing to an element with a volume, which would make it easier to interpret. A further development here would be the possibility to render and shade the drawings built in the model.
- Ensuring that the optimum performance is achieved from the software by changing some of the internal structures. *Quark* uses linked lists to maintain the models inside of the computers memory while the drawing is opened and active. These lists can be replaced by multi-threaded binary tree structures that would enhance the manipulation and search facilities of *Quark* considerably.
- Replacing the graphic display techniques with more industry specific methods like *OpenGL* or *Sprite* display libraries to ensure optimum performance for the graphical interface. All primitives and drawings currently in *Quark* make use of techniques developed specifically for the software.
- Providing users with the ability to fully manipulate the environment in which they work. *Quark* currently provides a user freedom inside the system but this should be extended by more environmental manipulation options and more user interaction in the form of data output. Data output can either be from the system, which is the graphical or pre-processor interface, or resulting from manipulation of the model inside of the graphical interface by the user.
- Maintaining a high level of interaction between the software and user is very important to ensure that users want to use the software and feel comfortable in doing so. Examples of this would be the animation of translation or rotation operations, the dynamic reporting of control parameters while executing and operation or the post-processor can be extended to 'loop back' to the pre-processor to feed the output data back to the input files for re-analysis.
- Adding colour coding to the display methods used in the post-processor to view the displacements or any other information calculated during the analysis process.

- Including a database interface into a database containing all the steel sections found in South Africa. This will facilitate the user tremendously when assigning cross section properties to elements in the structure.

These are a few of the possible improvements that can be made to increase the functionality of graphical interface.

4.5.2 The Analysis Interface

This is the most crucial part of the software and it also presents the greatest opportunity for improvement. The potential improvements include enhancements to the current algorithms and data structures as well as the addition of new algorithms.

The following enhancements can be made to the internal data structures used for analysis purposes:

- Stiffness matrices created can be written into banded form. This will reduce the amount of memory used to store these large volumes of data.
- The structures can be adjusted to automatically compress the data stored when entering and decompressing data when extracting. Dynamic data is maintained in the computer *RAM* memory. By adding the compression functionality better memory use will be ensured and the size of structures that can be analysed will increase accordingly.
- This functionality can be further extended by writing the compressed structures to disk and retrieved from the disk when extracting the data. This process will slow down the speed at which data can be manipulated but will considerably enhance the size of the models that can be analysed by the system. The method suggested compares to standard *paging* functions provided by most operating systems.

Adding the following can enhance the current structural analysis algorithms in the system:

- Coding the matrix and vector manipulation procedures in a lower level language like *Assembler* will increase the speed at which the procedures are executed.
- Better error handling and general recovery procedures will ensure that the system can perform better where resources are stretched to their limits.
- The general process of execution of the analysis algorithms can be changed to remove preparation and cleanup or recovery procedures out of the main execution loops. This will ensure that the analysis algorithms produce less overhead processing and lead to faster execution.

Additional algorithms for analysis will include the following:

- The current two dimensional plastic analysis algorithms can be extended to a three dimensional algorithm. This will mainly involve the enhancement of the structural theories to a three dimensional state. Only after completing this task can the computer algorithms be adjusted to analyse structures accordingly. This task in itself presents a topic for a complete thesis, or more.
- Inclined supports, spring supports, dynamic analysis and many more structural analysis procedures can be added to the current algorithms to enhance the functionality of what is currently provided. This topic once again is too broad to discuss any further at this point.

It is clear from these couple of suggestions that there is much room for future work and improvement to both the structural theories and the software.

CHAPTER 5

Conclusions

Structural mechanics were among the pioneers of finite element analysis methods, which is widely used in many disciplines. This status enjoyed by the structural profession has not changed much since the emerging years of the technology and hopefully will not change much in the future. This is ensured by the demands placed on the engineering profession, which uses computer systems to assist in the analysis of complicated problems not suited for classical solution.

Although much research have been done regarding these analysis methods since it's founding years, with ever increasing computer power at lower costs, there are new possibilities of study opening up every day. This thesis examined two main sections. Firstly, to extend the capabilities of internal variable formulation for plane frames to include second order effects and frame elements with semi-rigid connections. This would mean that the solution could be extended to include other problem domains and criteria to finally provide a better solution for a wider range of problems. Secondly, to investigate the use of software programming theories and the application thereof in the field of structural analysis. This topic was extended too include the structural analysis and user interface, the pre process to the analysis.

The results obtained using *Quark* verify that the accuracy of this method compares favourably with results of other, more established methods. The formulation provides a suitable method for analysing plane steel structures where the material behaviour can be elastic plastic with the postyield response ranging from softening to hardening in either rigid or semi-rigid element end connections. The ability to cope with perfectly plastic behaviour makes it a suitable method for implementation into programs linked to steel codes where collapse load factors are generally sought. Most of the examples concentrate on proportional loading but the incremental nature of the method makes it applicable to more general cases where the structural loads are applied in any sequence.

The solution procedures developed for computing the plastic hinge rotations are very efficient due to the fact that the formulation restricts the plastic behaviour of the structure to the possible hinge positions. With the formulation already extended to include truss elements and semi-rigid frame element end connections, the natural advances of the formulation will be to include axial and shear forces as well as extending the formulation to three dimensions. With this in mind *Quark* was designed and built with a three dimensional graphical interface. In order to accommodate the changes, additional internal variables are required at member ends and the relationship between them and their conjugate forces needs to be defined.

One of the main goals of designing *Quark* was to assist during the design, processing and interpretation stages of the analysis of a steel structure and to apply software design principles to the field of structural analysis. It has been proven in this study that the principles behind Object-Oriented Programming are ideally suited to the nature of structural analysis. The reasons being that structures, although containing many variables, are variables themselves. In other words, none of the quantities of elements, nodes etc. are fixed in structures, they range from small to extremely large. The second primary reason is that this method of programming provides

configuration flexibility only found when implementing this design principle. This is exactly the flexibility that is required when configuring and working with structural analysis problems.

The design method used, Object-Oriented Programming, ensures that computer resources are used at an optimum which translates into a direct saving of time when analysing the structures. The flexibility provided by the dynamic nature of the design method allows engineers to quickly reconfigure, add or delete elements to the structure. This is a great advantage over previous methods of user input, which ensures more accuracy and ensures a great saving of time.

Interpreting the output from the analysis of a model is the most important part of the analysis cycle. Without the correct interpretation of results and without accurate results any analysis method defeats the purpose of the exercise. *Quark* facilitates users in understanding the results by displaying the results in the form of a graphical output and by giving a user the ability to view or print the results in a grid format, which gives the user more defined information. The information is presented in an understandable and easy to access format, which in former analysis methods required a lot of time for interpretation and comprehension. It is thus safe to say that *Quark* contributes considerably to the overall usability, flexibility and interpretation of the structural analysis process and has achieved the goal set out. The functionality in *Quark* is limited to the structural analysis domain and has little or no use outside of this area, which was one of the original design principles.

There are many areas, which can be considered for improvement in *Quark* as mentioned in *Section 4.5*. It is important to remember that software is a tool that aids the analysis process and that *Quark* and the principles mastered during the design and implementation thereof was a starting point from which to build future projects. Analysis methods and software will continue to develop as requirements change at an increasing rate. Writing software for such specialised areas as structural analysis purposes should be limited to cases where commercial software does not provide for the analysis method or where new theories or procedures are being tested. The cost of writing custom software cannot be justified in a commercial environment when considering the time involved in writing and debugging software.

Unfortunately, in South Africa funds are not readily available for large software development programs at Universities to ensure the establishment and development of such technologies as will be required in the future. The technology and knowledge is, however, available and could be applied in the future. The combination of structural analysis with computer technology is here to stay and this in turn gives rise to the following concluding question: *"Where and how will the technology and requirements emerge and where will it be applied next?"*

Bibliography

Booch, G., *Object-Oriented Analysis and Design With Applications, 2nd Edition,* Benjamin Cummings, 1994

Calvert, C., *Delhi Unleashed,* SAMS Publishing, 1995

Chen, W.F., Toma, S., *Advanced Analysis of Steel Frames,* CRC Press, 1994

Cohn, M.Z., Erbatur, F. and Franchi, A. *STRUPL 1: User's Manual,* University of Waterloo Press, 1982

Cook, R.D., Malkus, D.S., Plesha, M.E., *Concepts and Applications of Finite Element Analysis, Third Edition,* John Wiley, 1989

Deitel, H.M., Deitel, P.J., *C++ How To Program,* Prentice Hall International, 1994

Martin, J.B., 'An internal variable approach to the formulation of finite element problems in plasticity', *in Physical nonlinearities in structural analysis* (Ed. J. Hult and J. Lemaitre), Springer-Verlag, 1980, pp. 165-176

NAG, *Linpack Routine Library,* 1994

Object-Oriented Programming, IEEE Expert, IEEE Computer Society, December 1990

Popov, E.P., *Engineering Mechanics of Solids,* Prentice Hall, 1990

Reiser, M., Wirth, N., *Programming in Oberon,* ACM Press, 1994

Roth, C.P., Bird, W.W., 'Internal variable formulation for the plastic analysis of plane frames', *Engineering Structures,* Vol. 17, No. 3, pp. 214 – 220, 1995

Rumbaugh, J., et al, *Object-Oriented Modeling and Design,* Prentice Hall, 1991

Shames, I.H., Dym, C.L., *Energy and Finite Element Methods in Structural Analysis,* Taylor & Francis, 1985

Van der Merwe, J., *Object Oriented Programming,* DataFusion Systems, 1996

Wozniewics, A.J., Shammas, N., *Tech Yourself Delphi in 21 Days*, SAMS Publishing, 1995